THE METAPRL LOGICAL PROGRAMMING ENVIRONMENT

Volume I

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Jason Jonathan Hickey

January 2001

© Jason Jonathan Hickey 2001
ALL RIGHTS RESERVED

THE METAPRL LOGICAL PROGRAMMING ENVIRONMENT

Jason Jonathan Hickey, Ph.D.
Cornell University 2001

This thesis is primarily about the design of formal programming environments for building large software systems. This work articulates two principles and uses them to guide the design, implementation, and study of a specific formal programming environment. First, design methods for large software systems will include multiple languages, methodologies, and refinement techniques that are suited to problem subdomains. This means that any formal system must provide the ability to define multiple logics, and it is by definition a *logical framework*. Second, the framework must provide the ability to express formal *relations* between logical theories to address the problem of system decomposition.

This thesis also presents the the MetaPRL formal system. MetaPRL was built to provide a modular, abstract logical framework where multiple designs can be expressed and related. The MetaPRL design builds on our experience with logical frameworks and with structured programming concepts like inheritance and re-use to provide an efficient, highly abstract, logical machine. The contribution includes several parts.

- The development of an untyped *meta*-logic using explicit substitution.
- The definition of a *very*-dependent function type in the Nuprl type theory.
- A system *architecture* for generic multi-logical development.
- A generic *refiner* that provides automation and enforcement for the multiple logical theories in logical environment.
- A *module system* for logics and theories.
- A generic *distributed interactive theorem prover*.

BIOGRAPHICAL SKETCH

Jason Jonathan Hickey was born in 1963 in a small town called Delano in the heart of California's central San Jaoquin valley. Jason's early experiences included the fulfillment of various agricultural obligations with his four sisters Elizabeth, Glynis, Heather, and Megan. Jason graduated as valedictorian from Delano High School in 1981. Content with his agricultural skills, Jason pursued a course of study in science and mathematics at the California Institute of Technology, where he graduated with honors and a Bachelor of Science degree in Electrical Engineering in 1985. It was during this time that he first met Takako Matoba, another Caltech student, who he would eventually marry.

During the next 10 years, Jason was employed as a Member of the Technical Staff at Bellcore, in Morristown New Jersey. His research at Bellcore centered around packet switch design, including architecture, VLSI design, and standards. In 1986, Jason attended Cornell University as part of Bellcore's graduate study program. He received a Master of Science degree in Electrical Engineering from Cornell in 1987. In 1992, Jason went back to Cornell as part of Bellcore's Doctoral Study Program, this time in Computer Science. During 1999-2000, Jason has been an Assistant Professor at the California Institute of Technology.

ACKNOWLEDGEMENTS

Life in graduate school has been an intensely personal, enriching experience filled with joy tempered by a struggle for understanding. Above all, I am grateful to Professor Robert Constable for the guidance and support he provided for issues both technical and personal. Bob has been a constant source of inspiration, while allowing me the freedom to pursue my goals and make my own mistakes. He has a wonderful ability both to shed light on the darkest times and celebrate the brightest.

I am also grateful to Aleksey Nogin, who has been both a friend and collaborator from the time when MetaPRL was still in its infancy. Aleksey has provided key theoretical insights that have sharpened our long-term focus, and he has also supplied the energy to guide the system through several major revisions to emerge in its current form as a high-performance logical programming environment.

I would like to thank Professors Dexter Kozen and Greg Morrisett. Dexter has given me a sense of the beauty of computer science, and Greg has helped broaden my perspective on programming languages and methodology. In addition, both have shown me that it is possible to have a satisfying life that includes research and teaching.

During the later stages of my studies, I worked in collaboration with Professor Nancy Lynch, Dr. Robbert van Renesse, Dr. Mark Hayden, and Dr. Mark Bickford on verification and optimization of the Ensemble system. This was a rewarding collaboration, and I would like to thank my collaborators for making it enjoyable.

Another immensely enriching experience at Cornell was doing a minor in sculpture. Since leaving home, I had always neglected the non-technical side of my life. Sculpture gave me a way of expressing more personal feelings in a tangible way. This experience changed my life, and I would like to thank Professor Gail Scott White for her energy and support and Professor Robert Bertoia for his encouragment and criticism.

Along the way, I would like to thank several friends for helping make life in Ithaca fun: Gary Adams, Eli Barzilay, Aswin van den Berg, Karl Crary, David Walker, and many others.

Finally, my wife Takako Hickey has been my friend throughout this process, and to her I am eternally grateful.

To Doris Scott Hickey

TABLE OF CONTENTS

I Introduction 1

1 Introduction 2
- 1.1 Methods for large software systems . . . 3
- 1.2 Contributions . . . 4
- 1.3 Some history and related work . . . 5
- 1.4 Outline of the thesis . . . 8

2 Basic Logical Foundations 9
- 2.1 Defining a simple logic of programs . . . 10
- 2.2 Defining additional programming operations . . . 13
- 2.3 Defining a type theory for the simple programs . . . 13
- 2.4 Defining a logic . . . 20
- 2.5 Proof automation . . . 24

3 Reasoning about programs 29
- 3.1 Defining an imperative language . . . 30
- 3.2 Deriving a type system . . . 34
  - 3.2.1 Developing the Boolean values . . . 35
  - 3.2.2 Interpreting locations and states . . . 37
  - 3.2.3 Interpreting the programs . . . 39
  - 3.2.4 Deriving the natural semantics . . . 39
  - 3.2.5 Enhancing the type theory . . . 42

4 Modular composition of reactive systems 45
- 4.1 I/O automata . . . 46
- 4.2 Stylized automata . . . 48
- 4.3 Type-theoretic interpretation . . . 49
- 4.4 Using the automata . . . 50
- 4.5 Operations on automata . . . 51
- 4.6 Specifying Ensemble Virtual Synchrony . . . 53
- 4.7 Related Work . . . 55

II The MetaPRL Logical Framework **57**

5 The MetaPRL Meta-Language and Logic **58**
5.1 The term language 58
5.1.1 Syntax definitions 61
5.2 The frame 66
5.2.1 The meta-logic 70
5.2.2 The sequent logic 70
5.3 Related Work 72

III The MetaPRL System **75**

6 The MetaPRL System Architecture **76**

7 The MetaPRL Module System **83**
7.1 The MetaPRL module system 84
7.2 Term syntax and the `declare` statement 84
7.3 Display forms 88
7.3.1 Options 92
7.3.2 The `slot` expression 92
7.3.3 Literals 92
7.3.4 Line breaking and zones 93
7.3.5 Indentation 93
7.3.6 Example 94
7.4 MetaPRL Axioms and Inference Rules 95
7.4.1 Signatures 95
7.4.2 Implementation (structures) 96
7.4.3 Tactics 97
7.5 MetaPRL computational rewriting 97
7.5.1 Implementations 98
7.6 Rule definitions and proofs 99
7.7 Theory inheritance 100
7.7.1 Naming conflicts 102
7.8 Rules, tactics, and resources 103
7.9 The MetaPRL compiler and logical library 105

8 The MetaPRL Refiner **108**
8.1 Introduction 108
8.2 Architectural Overview 109
8.3 The *term* module 112
8.3.1 Naive term implementation (`Term_std`) 112
8.3.2 Delayed substitution (`Term_ds`) 114
8.4 The *rewriter* module 117

8.5 The *proof* module 120
8.6 Performance 120
8.7 Related work 123

9 The MetaPRL Distributed Prover 125
9.1 Introduction 125
9.2 Refinement Architecture and Tactics 126
9.3 Multithreaded Refinement 129
9.3.1 Proof tree data type for concurrent processes . . . 129
9.3.2 Threaded Refinement 131
9.4 Distributed refinement 133
9.4.1 Distributed Memory Architecture 135
9.4.2 Distributed Refinement 136
9.5 Performance Measurements 137
9.6 Summary, implementation notes, and future work 139
9.7 Related Work 139

IV Theoretical Issues and Extensions 141

10 Modular Theories, and the "Very-Dependent" Function Type 143
10.1 Introduction 143
10.2 Module example 145
10.3 Very-dependent type extension 147
10.3.1 Very-dependent function type 151
10.3.2 Very dependent recursive types 156
10.4 A Theory of Objects 157
10.4.1 Formal components of an "object–system" 159
10.4.2 Polymorphism and record extension 161
10.4.3 Propositional meaning of signatures 164
10.4.4 A record calculus 165
10.4.5 Object system 165
10.4.6 Types for an object system 166
10.5 Type theory and "theories" 169
10.5.1 Type theory foundations 170
10.5.2 Sequent templates and proofs 173
10.5.3 Objects of the type theory 175
10.6 Related-work 176

11 A Predicative Interpretation of Objects 178
11.1 Introduction 178
11.2 Review 180
11.2.1 Record encoding 181

11.3 Object Interpretation 182
11.3.1 Constrained polymorphism 183
11.3.2 Non-recursive interpretation 184
11.3.3 Recursive Objects 185
11.4 Dependent Objects 190
11.4.1 Encoding 192
11.4.2 Dependent subtyping: $\preceq^{dep}$ 193
11.5 Related Work 194

Volume II **195**

V Logics Implemented in MetaPRL **196**

12 The MetaPRL Logics **197**
12.1 Base_theory module 198
12.1.1 Parents 198
12.2 Summary module 199
12.2.1 Terms 199
12.3 Mptop module 202
12.3.1 Parents 202
12.4 Var module 203
12.4.1 Parents 204
12.5 Top_tacticals module 204
12.5.1 Parents 204
12.5.2 Primitive tactics 204
12.5.3 Tacticals 205
12.5.4 Tactic arguments 206
12.5.5 Clause selection 207
12.5.6 Labels 207
12.6 Top_conversionals module 208
12.6.1 Parents 208
12.6.2 Conversion application 209
12.6.3 Primitive conversions 209
12.6.4 Conversionals 209
12.6.5 Addressing and search 209
12.6.6 Conversion reversal 210
12.6.7 Resources 211
12.7 Base_trivial module 211
12.7.1 Parents 212
12.8 Base_auto_tactic module 212
12.8.1 Parents 213

12.9 Base_dtactic module 213
12.9.1 Parents 215
12.10 Base_rewrite module 215
12.10.1 Parents 215
12.10.2 Rules 215

13 First-Order Logic **217**
13.1 Defining a First-Order Logic 217
13.1.1 Getting started 218
13.1.2 Defining well-formed formulas 219
13.1.3 Logical constants 222
13.1.4 Basic Automation 224
13.1.5 Adding the propositional connectives 229
13.1.6 Defining negation 232
13.1.7 Adding structural rules 245
13.1.8 Defining a propositional logic 247
13.1.9 Proof automation for constructive propositional logic 251
13.1.10 The quantifiers 259
13.2 Relating the propositional logic to type theory 260
13.3 Defining classical first-order logic 265
13.4 Deriving classical first-order logic from the Nuprl type theory 268
13.5 Conclusion 273

14 The Nuprl type theory **274**
14.1 Overview of the Nuprl type theory 275
14.1.1 Nuprl proof system 277
14.2 Itt_squash module 282
14.2.1 Parents 283
14.2.2 Resources 283
14.2.3 Tactics 283
14.3 Itt_equal module 283
14.3.1 Parents 283
14.3.2 Terms 283
14.3.3 Rules 284
14.4 Itt_struct module 287
14.4.1 Parents 287
14.4.2 Terms 288
14.4.3 Rules 288
14.4.4 Tactics 290
14.4.5 Resources 291
14.5 Itt_subtype module 292

14.5.1 Parents . . . 292
14.5.2 Terms . . . 292
14.5.3 Rules . . . 292
14.5.4 Resources . . . 294
14.6 Itt_void module . . . 294
14.6.1 Parents . . . 294
14.6.2 Rules . . . 295
14.7 Itt_unit module . . . 295
14.7.1 Parents . . . 296
14.7.2 Rules . . . 296
14.8 Itt_atom module . . . 297
14.8.1 Parents . . . 297
14.8.2 Terms . . . 298
14.8.3 Rules . . . 298
14.9 Itt_set module . . . 299
14.9.1 Parents . . . 299
14.9.2 Terms . . . 299
14.9.3 Rules . . . 299
14.10 Itt_union module . . . 301
14.10.1 Parents . . . 302
14.10.2 Terms . . . 302
14.10.3 Rewrites . . . 302
14.10.4 Rules . . . 302
14.11 Itt_rfun module . . . 305
14.11.1 Parents . . . 305
14.11.2 Terms . . . 305
14.11.3 Rewrites . . . 306
14.12 Itt_dfun module . . . 311
14.12.1 Parents . . . 311
14.12.2 Rewrites . . . 311
14.12.3 Rules . . . 312
14.13 Itt_fun module . . . 314
14.13.1 Parents . . . 314
14.13.2 Rewrites . . . 314
14.14 Itt_well_founded module . . . 316
14.14.1 Parents . . . 316
14.14.2 Terms . . . 317
14.14.3 Rewrites . . . 317
14.14.4 Rules . . . 317
14.15 Itt_dprod module . . . 318
14.15.1 Parents . . . 319
14.15.2 Terms . . . 320
14.15.3 Rewrites . . . 320

14.15.4 Rules 320
14.16 Itt_prod module 323
14.16.1 Parents 323
14.16.2 Rewrites 323
14.16.3 Rules 323
14.17 Itt_logic module 325
14.17.1 Parents 325
14.17.2 Terms 325
14.17.3 Rewrites 325
14.17.4 Rules 326
14.17.5 Tactics 332
14.18 Itt_decidable module 335
14.18.1 Parents 335
14.18.2 Terms 335
14.18.3 Rewrites 335
14.18.4 Rules 335
14.18.5 Resources 335
14.18.6 Tactics 336
14.19 Itt_bool module 336
14.19.1 Parents 336
14.19.2 Terms 337
14.19.3 Rewrites 337
14.19.4 Rules 338
14.19.5 Tactics 343
14.20 Itt_int module 343
14.20.1 Parents 343
14.20.2 Terms 344
14.20.3 Rewrites 344
14.20.4 Rules 345
14.21 Itt_isect module 347
14.21.1 Parents 348
14.21.2 Terms 348
14.21.3 Rules 348
14.22 Itt_bisect module 351
14.22.1 Parents 351
14.22.2 Terms 351
14.22.3 Rewrites 351
14.22.4 Rules 351
14.23 Itt_tunion module 353
14.23.1 Parents 353
14.23.2 Terms 353
14.23.3 Rules 353
14.24 Itt_bunion module 355

14.24.1 Parents . . . 355
14.24.2 Terms . . . 355
14.24.3 Rewrites . . . 355
14.24.4 Rules . . . 355
14.25 Itt_w module . . . 356
14.25.1 Parents . . . 357
14.25.2 Terms . . . 357
14.25.3 Rewrites . . . 357
14.25.4 Rules . . . 358
14.26 Itt_prec module . . . 359
14.26.1 Parents . . . 360
14.26.2 Terms . . . 360
14.26.3 Rewrites . . . 360
14.26.4 Rules . . . 361
14.27 Itt_srec module . . . 363
14.27.1 Parents . . . 363
14.27.2 Terms . . . 363
14.27.3 Rewrites . . . 364
14.27.4 Rules . . . 364
14.28 Itt_list module . . . 366
14.28.1 Parents . . . 366
14.28.2 Terms . . . 366
14.28.3 Rewrites . . . 367
14.28.4 Rules . . . 367
14.29 Itt_list2 module . . . 370
14.29.1 Parents . . . 370
14.29.2 Terms . . . 370
14.29.3 Rewrites . . . 371
14.29.4 Rules . . . 375
14.30 Itt_quotient module . . . 377
14.30.1 Parents . . . 378
14.30.2 Terms . . . 378
14.30.3 Rules . . . 378
14.31 Itt_esquash module . . . 382
14.31.1 Parents . . . 383
14.31.2 Terms . . . 383
14.31.3 Rewrites . . . 383
14.31.4 Rules . . . 383
14.31.5 Tactics . . . 385
14.32 Itt_theory module . . . 386
14.32.1 Parents . . . 386

15 Aczel's CZF Set Theory **387**
15.1 Czf_itt_theory module 388
15.1.1 Axioms 388
15.1.2 Theory organization 390
15.1.3 Parents 391
15.2 Czf_itt_set module 391
15.2.1 Parents 393
15.2.2 Terms 393
15.2.3 Rewrites 393
15.2.4 Rules 394
15.3 Czf_itt_eq module 395
15.3.1 Parents 396
15.3.2 Terms 396
15.3.3 Rewrites 396
15.3.4 Rules 398
15.3.5 Tactics 400
15.4 Czf_itt_member module 401
15.4.1 Parents 401
15.4.2 Terms 401
15.4.3 Rewrites 401
15.4.4 Rules 401
15.4.5 Tactics 403
15.5 Czf_itt_fol module 403
15.6 Czf_itt_dall module 404
15.6.1 Parents 405
15.6.2 Terms 405
15.6.3 Rewrites 405
15.6.4 Rules 405
15.7 Czf_itt_dexists module 406
15.7.1 Parents 407
15.7.2 Terms 407
15.7.3 Rewrites 407
15.7.4 Rules 407
15.8 Czf_itt_sall module 409
15.8.1 Parents 409
15.8.2 Terms 409
15.8.3 Rewrites 409
15.8.4 Rules 409
15.9 Czf_itt_sexists module 410
15.9.1 Parents 410
15.9.2 Terms 410
15.9.3 Rewrites 410
15.9.4 Rules 411

15.10 Czf_itt_subset module 411
15.10.1 Parents 411
15.10.2 Terms 412
15.10.3 Rewrites 412
15.10.4 Rules 412
15.11 Czf_itt_sep module 413
15.11.1 Parents 413
15.11.2 Terms 413
15.11.3 Rewrites 414
15.12 Czf_itt_empty module 416
15.12.1 Parents 416
15.12.2 Terms 416
15.12.3 Rewrites 416
15.12.4 Rules 416
15.13 Czf_itt_singleton module 416
15.13.1 Parents 417
15.13.2 Terms 417
15.13.3 Rewrites 417
15.13.4 Rules 417
15.14 Czf_itt_union module 418
15.14.1 Parents 418
15.14.2 Terms 418
15.14.3 Rewrites 419
15.14.4 Rules 419
15.15 Czf_itt_pair module 421
15.15.1 Parents 421
15.15.2 Terms 421
15.15.3 Rewrites 421
15.15.4 Rules 422
15.16 Czf_itt_nat module 423
15.16.1 Parents 423
15.16.2 Terms 423
15.16.3 Rewrites 424
15.16.4 Rules 424
15.16.5 Tactics 426
15.17 Czf_itt_rel module 426
15.17.1 Parents 426
15.17.2 Terms 427
15.17.3 Rewrites 427
15.17.4 Rules 427
15.18 Czf_itt_power module 427
15.18.1 Parents 428
15.18.2 Terms 428

15.18.3 Rewrites 428
15.18.4 Rules 428
15.18.5 Tactics 429
15.19 Czf_itt_axioms module 429
15.19.1 Parents 429
15.19.2 Rules 430

BIBLIOGRAPHY **432**

LIST OF TABLES

2.1 Syntax for a simple type theory 11
2.2 Operational rules for the simple type theory 11
2.3 Interpretation of the syntax of programs in the simple type theory 12
2.4 Interpretation of the operational rules in the simple type theory 12
2.5 Deriving the standard fixpoint reduction 14
2.6 Some standard arithmetic operations in the simple type theory 16
2.7 The syntax of types for the simple type theory 16
2.8 Typehood judgments for the simple type theory 17
2.9 Interpretation of types in the simple type theory 18
2.10 Membership introduction rules in the simple type theory . 19
2.11 Structural and elimination rules in the simple type theory 19
2.12 Typing of addition in the simple type theory 20
2.13 Interpretation of propositions as type 21
2.14 Predicate rules in the simple type theory 22
2.15 Derivation of and–introduction 23
2.16 Refiner operations 26
2.17 Tactic primitives 26
2.18 Proof procedure for propositional logic 27
2.19 Tactic implementations 28

3.1 Syntax of simple imperative programs 30
3.2 State operations for the simple imperative language 30
3.3 Operational description of state operations 31
3.4 Definition of values for simple imperative programs 31
3.5 Natural semantics for simple imperative programs 32
3.6 Encoding of state operations 33
3.7 Defining a version of Boolean values 36
3.8 Interpretation of states and locations 38
3.9 Program interpretations 40
3.10 Evaluation of application 42
3.11 Interpretation of records in STT 43

3.12 Additional judgments in the type theory 44

5.1 Rules of the meta-logic 71
5.2 Rules of the propositional logic 72
5.3 Part of the proof of $A \Rightarrow ((A \Rightarrow \bot) \Rightarrow \bot)$ 73

7.1 Syntax of formal signatures 85
7.2 Term syntax 86
7.3 Simplified syntax for logical operators 89
7.4 Display forms for a fragment of first-order logic 91
7.5 A complex display form 94

8.1 Term module signature 113
8.2 Substitution for Term_std 113
8.3 Terms with delayed substitution 115
8.4 Free variable calculation 116
8.5 Delayed substitution algorithm 116
8.6 Substitution expansion for Term_ds 117
8.7 Term expansion for Term_ds 117
8.8 Signature for the rewrite module 118
8.9 Virtual machine instructions 120
8.10 The proof module signature 121

9.1 Proof procedure for propositional logic 128
9.2 Tactic module signature 129
9.3 Scheduler implementation 134

10.1 Signature function 168

13.1 Listing of the `theories` directory 218
13.2 Boilerplate `Makefile` code 220
13.3 Building MetaPRL 220
13.4 Editor listing for the `fol_type` module 222
13.5 `Fol_and` module for conjuction 233
13.6 `Fol_or` module for disjunction (part 1) 234
13.7 `Fol_or` module for disjunction (part 2) 235
13.8 Listing of the `Fol_not` module 238
13.9 Proving the `not_type` rule (steps 2, 3) 239
13.10 Proving the `not_type` rule (steps 4, 5) 240
13.11 Proving the `not_type` rule (step 6) 241
13.12 Proof of the `not_intro` rule 242
13.13 Prove the `not_elim` rule (step 1) 243
13.14 Prove the `not_elim` rule (step 2) 244
13.15 Prove the `not_elim` rule (step 3) 244
13.16 Finish the `fol_not` module 245

13.17 Structural rules 246
13.18 Prove the distrib_or theorem 249
13.19 Proof steps taken by autoT 250
13.20 Prove the distrib_or theorem 252
13.21 Rules for implication elimination 254
13.22 Tactics for implication elimination 255
13.23 Term predicate functions for testing implication 256
13.24 Proof search algorithm 257
13.25 Applying the propDecideT tactic to distrib_or 258
13.26 A simple tactic for proving well-formedness subgoals . . 258
13.27 The Fol_pred module 259
13.28 Definition of the universal quantifier 261
13.29 The first-order logic Fol_theory 262
13.30 The interpretation of the false constant 263
13.31 Proving the false_type rule 264
13.32 Constructive interpretation of the logical operators 264
13.33 Listing of the Fol_itt_and module 266
13.34 Prove the and_type theorem 267
13.35 Providing classical reasoning 268
13.36 Base interpretation Cfol_itt_base (Part 1) 270
13.37 Base interpretation Cfol_itt_base (Part 2) 271
13.38 Base interpretation Cfol_itt_base (Part 3) 272
13.39 Interpretation of the classical operators 272
13.40 Prove the all_elim theorem 273

LIST OF FIGURES

2.1 Example propositional proof 25
2.2 Tactic-prover architecture 25

5.1 Hierarchy of languages 59

6.1 Typical workflow 78
6.2 Component architecture 81

7.1 Multiple theory inheritance 100
7.2 A “Set” resource 104
7.3 An example of compiling first-order logic 106

8.1 General tactic prover architecture 109
8.2 Layered refiner 111
8.3 Rewrite virtual machine 119

9.1 Example propositional proof 126
9.2 Tactic-prover architecture 127
9.3 A partial proof tree and a proof scheduler 130
9.4 Proof tree operations 131
9.5 Scheduling operations 132
9.6 Distributed refiner block diagram 137
9.7 Performance on the five domains 138

10.1 Telescope 149
10.2 Tree of dependent types 150
10.3 Rules for dependent calculus 158
10.4 Rules for nondependent calculus 162

11.1 Syntax of the object calculus 180
11.2 Rules for nondependent calculus 186
11.3 Rules for dependent calculus 190

Part I

Introduction

Chapter 1

Introduction

In 1969, Joseph Bates and Robert Constable joined forces to produce the first PRL ("Program Refinement Logic") methodology, combining the use of constructive mathematics as a high-level programming language and logic [Con71, CO78] with the use of refinement as a design model for verifiable software development [Bat79]. In the PRL methodology, programming and verification were indistinguishable activities, and the idea of programming became a methodical process of discrete, manageable design decisions. The first system to implement the methodology was *lambda*-prl [BC85, Sta83], which was released in 1983.

At the time, software systems were undergoing tremendous stresses. The development of the integrated circuit in the 1960's led to consistent exponential growth in hardware resources, and software designers were hard pressed to find the structure and methodology that would lead to manageable, reliable systems. The PRL system proposed a fundamental change to the design process. Instead of the normal process, where programming was the primary task followed by testing and verification, the PRL system presented a unified approach, where a program would be *synthesized* by a sequence of refinements of the software objective (or specification). Logic was used not only for the semantics of the programming language, but also to formalize (and then automate) the design methodology.

Presently, at the turn of 21st century, the rate of growth in silicon-based computing has continued. Personal computers now have more than a thousand times the power of a machine in 1980 and routinely run programs consisting of tens of millions of lines of code written by teams of thousands of programmers.

During the last 20 years, the PRL group has continued to release larger and more comprehensive formal programming environments. The Nuprl 1 system [CAB+86], released in 1984, introduced a rich type structure, based on Martin–Löf type theory, that joined the no-

tion of propositions and types, and introduced explicit treatments of equality and program simplification through the use of computational and equality rules. A major effort in subsequent versions, starting with Nuprl 3, has been to substantially enrich the automation framework. Nuprl 4 [Jac94] introduced an extensive tactic library for general-purpose reasoning as well as a powerful conversion library for computation and term rewriting. The goal in Nuprl 5 [ACE+00] has been to support general collaborative development using multiple theorem provers in a heterogeneous logical programming environment.

In the broader context, the formal community as a whole has developed *logical frameworks* that permit the definition of multiple logics united in a common automation platform, most significantly with the Isabelle [NP92] theorem prover and Edinburgh Logical Framework [HHP93, Pfe94]. At this point in time, the goals posed by Constable and Bates have been partially achieved. The formal methods community is adept at developing individual algorithms by refinement, and there have been several examples of reasonably large formal projects, including procotol verification for Ensemble [Kre97, Kre99]; Jackson's formalization of a large part of constructive abstract algebra [Jac95]; the formalization of the Java semantics in Isabelle [NvO98] as well the formalization of I/O automata in Isabelle/HOL [MN95]; and others.

Yet, while the progress in formal development has been substantial, it is still not enough to address the exponential growth in the software industry. The industrial model has not altered; program development remains a task of programming followed by extensive testing. Part of the problem is the lack of adequate automation to make refinement competetive with existing technology. A more fundamental issue is that real world problems are often highly heterogeneous, requiring several design techniques to address different aspects of a problem.

1.1 Methods for large software systems

The work presented here is based on two essential concepts. First, design methods for large software systems must include multiple languages, methodologies, and refinement techniques that are suited to problem subdomains. This means that any formal system must provide the ability to define multiple logics, and it is by definition a *logical framework*. Second, the framework must provide the ability to express formal *relations* between logical theories to address the problem of system decomposition.

This thesis is primarily about the *design* of the MetaPRL formal system. The MetaPRL design goal has been to provide a modular, abstract

logical framework where multiple design methods (including the Nuprl type theory) can be expressed and related. The MetaPRL design builds on our experience with logical framework and with structured programming concepts like inheritance and re-use to provide an efficient, highly abstract, logical machine. At the time of writing, the MetaPRL system is in public release as version 0.6, and has attracted a range of users/developers from around the world.

1.2 Contributions

The contributions of this thesis can be divided into two parts: system development and engineering, and theoretical foundations needed to support the effort. On the theoretical side, there are two major parts.

- The development of an untyped *meta*-logic using explicit substitution (Chapter 5). The meta-logic differs from other meta-logics, such as the meta-logics in Isabelle and ELF, because substitution has explicit forms for expressing *computational equivalences* that differ from the forms for *logical inference rules* and *axioms*.
- The definition of a *very*-dependent function type in the Nuprl type theory (Chapter 10). The very-dependent function type provides the *logical* specification of theories and modules, and it provides the basis for compositional reasoning.

The MetaPRL system is the culmination of many years of effort. I have tried to keep the description concise, but it has eluded me. The system itself has several design concepts.

- A system *architecture* for generic multi-logical development (Chapter 6). The architecture ties together a *refiner* (the theorem prover in a PRL system), a logical *library* of system components, an interactive proof system, and a compiler.
- A generic *refiner* that provides automation and enforcement for the multiple logical theories in logical environment (Chapter 8). The refiner provides the basis for defining and relating logics, and it is also critical performance bottleneck. We have been able to keep the refiner generic but efficient.
- A *module system* for logics and theories (Chapter 7). The module system provides an inheritance mechanism that is the basis for *re-use* of proofs and programs as well as for expressing *relations* between theories.

- A generic *distributed interactive theorem prover* (Chapter 9). The distribution mechanism is tactic-based, interactive, and multi-logical. System faults, such as machine and network failures, are handled transparently, allowing a developer to make maximum use of the available computational resources for development.
- A demonstration that the system design is fertile and sound, with performance results, design milestones, the creation of a MetaPRL community, and the integration of results.

In addition, the MetaPRL system includes several logics.

- A reformulation of the Nuprl *type theory* (Chapter 14) expressed as a modular logic. The modularity provides partitioning of the Nuprl subtheories, and theory *relations* are used to derive many of the standard types from more expressive type constructors (reducing the total number of type theory primitives).
- A formulation of pure *first-order logic* (Chapter 13), with constructive and classical versions. The classical first-order logic is expressed as a *subtheory* of the constructive theory, and both versions are *related* to the Nuprl type theory through a type-theoretic encoding.
- A formulation of Aczel's Constructive ZF set theory (Chapter 15). This theory demonstrates the construction of a logic substantially different from the other, more type-theoretic, logics. In addition, the CZF theory includes a type-theoretic encoding that *relates* it to the Nuprl type theory. The relation is complete, meaning that theorems in the CZF theory can be used verbatim within the Nuprl type theory.

1.3 Some history and related work

From 1985-1992, I worked at Bellcore, in Morristown NJ, as a hardware architect and VLSI designer building high-speed packet switches. The usual development process in our research group was to work together to produce a high-level design, then partition the development according to our strengths, and start the development cycle. The initial stages held optimism and joy, which was much needed as we entered the long development process that would bring our ideas to fruition. We knew that the VLSI design would be an intensive effort, alternating between actual design and software support that would help coax the CAD tools to do our bidding.

When I entered Cornell University in 1992, I held goals born out of frustration with the design process. I felt that too much design time was spent turning the development crank, and the real culprit was design automation. Why not provide the outline of a design (the fun part), and have a machine produce the finished product? The PRL mission of program development by refinement seemed an ideal fit, and with Constable's encouragement, I set to work.

As an initial test case, Constable proposed a formalization of the *majority vote algorithm*: given a bag of size n, and a number k, find the elements in the bag that occur k or more times. The algorithm can be generated from a high-level $\forall\exists$ proof: for all bags and majorities k, there is a "reduced" bag that contains all the majority elements. David Gries had produced a concise two-page paper proof that produced an elegant algorithm. In the PRL group, Doug Howe and Chetan Murthy had independently formalized and derived the algorithm, but the elegance of Gries's solution was lacking. Thus began my initiation. The abstract data type for a bag could be formalized as a dependent product with a carrier, a set of methods, and a few axioms. The voting algorithm would follow from an inductive argument. That turned out to be true, but the real heart of the problem lay in producing a working multiset. The methods and their axioms are highly related, and a false step in one part can cause a lot of wasted effort before the flaw is discovered.

This seemed to be a general problem, and I set out in search of a general solution. The main problem appeared to be a lack of modularity that would permit an incremental development process. Perhaps we could produce a library of general principles and methods that would allow 1) the definition of a formal "object", and 2) re-use of proofs with a formal principle for "inheritance." Motivated by the idea of theories-as-classes, I set upon a formalization of classes and objects in the Nuprl type theory as dependent, extensible products. General recursion would be used to provide self-reference in the theory/class. Of course, the use of general recursion would also require a single, simultaneous, proof of well-formedness and termination for the entire mechanism. Needless to say, I got bogged down, and eventually exhausted the computational resources in our department.

To me, it seemed the idea of the proof was obvious; it just hard to prove. Instead, I next proposed that we add a *primitive* type construction to the Nuprl type theory that would avoid the need for an all-out justification. This lead to the development of the "very"-dependent function type (discussed in Chapter 10). The type allows type dependencies between the different *values* of a function. So, for instance, a function f the proof of a DeBruijn telescope $x_1{:}T_1, \ldots, x_n{:}T_n(x_1, \ldots, x_{n-1})$ if $f(x_i) \in T_i(f(x_1), \ldots, f(x_{i-1}))$. In addition, the type allows the formu-

lation of *dependent record* types, which are the principle components of formal theories/classes.

This was a step forward, but there were still problems. The proofs of typehood were still difficult, monolithic, arguments. The type provided a logical foundation, but a method for automation was still missing. This led to my first thoughts on MetaPRL. My initial goal had been to provide an automated framework for reasoning about and re-using formal abstract data types within the Nuprl type theory. An ADT can be viewed as a *module* or *theory* that contains a variety of fields, including values, methods, axioms, and proof automation code. It seemed obvious, why not generalize the type theory as well? This led to the general concept of the system as a *logical framework*.

Another issue with Nuprl at the time was that the Nuprl system was expressive, but it was also cumbersome and slow. The hope was, by re-implementing Nuprl as a small, modular system, we could improve the formal design process as well as achieve major performance gains. Thus the Nuprl-Light system was born. My goal was to combine ideas from logical frameworks, like Isabelle [NP92, Pau90] and LF [HHP93], with a modern modular programming language, specifically Caml-Light [Ler93] (hence the name).

The Isabelle and LF meta-logics were not immediately suited to formalizing the Nuprl type theory. First, there is a syntactic difference: the Nuprl type theory uses a general syntax that provides a general binding mechanism. Second, programs in a Nuprl system are *untyped*: a program may have multiple types that depend on the *context* in which the program is used. Third, the Nuprl logic is most easily formalized using explicit substitution in two forms: capturing substitution for computational rewriting, and capture-avoiding substitution for inference rules. Neither of these are theoretical obstacles. Isabelle uses hereditary Harrop formulas with logic variables as its meta-logic, and LF uses a higher-order constructive type theory. It is possible to construct the Nuprl type theory in both systems, although a mechanism for efficient computational rewriting would have to be devised. Finally, I felt that neither system provided an adequate account of "programming in the large," where the emphasis is on *relations* between programs, languages, and logics. In the end, I borrowed the meta-logical concept from Isabelle, modified as untyped second-order Horn clauses over Nuprl terms. The outer structure of the systems is quite similar.

Over time, the Nuprl-Light system grew to include a term rewriting engine, a refiner (the theorem prover), a logical library, a proof editor, and a complete modular implementation of the Nuprl type theory. In 1996, Alexey Nogin joined the project, and with his help we reached our performance objectives. Nuprl-Light began to outperform

the Nuprl system by a factor of more than 100. At the same time, I began to explore the formal analysis and construction of the Ensemble [Hay97, Hay98] distributed communication protocols, and I added a general-purpose distributed proving mechanism to Nuprl-Light. By 1998, a team of faculty and students at Moscow State University had joined the project, and our work had begun to attract interest on the Internet. We added additional logics, including first-order logic and constructive set theory, as well as a preliminary formalization of LF. Finally, Nuprl-Light began to provide functionality comptetive with Nuprl, and we rechristened it with the new name MetaPRL.

1.4 Outline of the thesis

The remainder of this thesis is divided roughly as follows. In keeping with the style of this introduction the next few sections give a broader overview of the process of formalizing and verifying a system from the ground up. This process goes through several steps listed below:

- The basic meta-logic that provides the basis for all logical development (Chapter 2).
- Defining a simple typed logic of programming (Section 2.1).
- Proof automation, and the MetaPRL *logical programming environment* (Chapter 2.5).
- Deriving a simple imperative language (Chapter 3) with a natural semantics (Section 3.1).

These sections are intended to give an overview of the contents and purpose of the material of this thesis without an overwhelming amount of technical detail.

The remainder of the thesis is divided into four parts that provide the technical justification, or simply expand on issues that were glossed over in this introduction. The first part discusses the logical foundations of the framework, including the meta-logic, and the primitives of the module system. The second part discusses the system implementation, its architecture, the theorem prover, and the module editor. The third part brings these two topics together with the exposition of several logics, including first-order logic, the Nuprl type theory, and the CZF set theory. Lastly, the fourth part discusses the remaining theoretical issues. Work on the logical framework is far from being done, and there remain basic semantic foundations for relating logics and programs.

Chapter 2

Basic Logical Foundations

If we are to enable automation in the development of systems, the parts to be automated must be *formal*. That is, they must be expressed in a language where statements, including programs and specifications, can be manipulated by algorithms that are executed on a machine. Since the task of verification is to relate programs and specifications, the formal language is by definition a logic.

Formal systems that support multiple logics are called *logical-frameworks*. The MetaPRL is a logical framework that emphasizes *relations* between logics for different domains. The MetaPRL design principle is this:

> **Any framework for large system development must be able to support multiple programming languages and specifications, and it must allow the parts to be related so that they can be composed.**

From a practical perspective, the system framework also needs an editor and compiler (so that program code can be created, modified, checked, and executed). Furthermore, the task of verification is expensive, and any practical system must allow verification *assumptions* that parts of the system behave as specified, and these assumptions must be open to scrutiny as necessary.

The framework architecture we propose takes as its foundation a meta-logic, and a method for automating proofs in the meta-logic. All statements, whether programs or specification, must be reducible to statements in the meta-logic. The MetaPRL logic is an unsorted second-order logic of terms, discussed in detail in Chapter 5. The basic syntax includes variables, second-order functions, and terms, defined inductively.

Variables:	v	::=	$v_1, v_2, \ldots$ (a countable collection)
Second-order applications:	V	::=	$V_1[\vec{t_1}], V_2[\vec{t_2}], \ldots$
Bound terms	bt	::=	$v_1, v_2, \ldots, v_n.t$
Terms	t	::=	$name(\vec{bt}) \mid V[\vec{t}]$

The variables in a bound term are binding occurrences (variables $v_1, v_2, \ldots, v_n$ are *bound* in the term t). *Statements* in the meta-logic are closed, quantified implications of the form:

$$\forall V_1, V_2, \ldots, V_n.t_1 \rightarrow t_2 \rightarrow \cdots \rightarrow t_m.$$

We usually omit the quantifiers for second-order variables, and consider all free second-order variables in a sentence $t_1 \rightarrow t_2 \rightarrow \cdots \rightarrow t_n$ to be implicitly universally quantified.

Meta-logical *proofs* are consequences: if Σ is a set of statements, and σ is a single statement, then $\Sigma \vDash \sigma$ is a meta-sequent. The sequent is *provable* if can be derived using two rules. The first rule is a derivation by assumption:

$$\Sigma \vDash t_1 \rightarrow \cdots \rightarrow t_i \rightarrow \cdots \rightarrow t_n \rightarrow t_i$$

There is only one other rule, to characterize *instantiation* of an assumption, where second-order variables are treated as functions over terms. If V is a second-order variable, and f is a function $term^n \rightarrow term$, we can instantiate a quantification of the form $\forall V.S$ by replacing all occurrences of $V[t_1; \ldots t_n]$ in S by the term $f(t_1, \ldots, t_n)$.

If $\sigma = \forall V_1, V_2, \ldots, V_p.s_1 \rightarrow \cdots \rightarrow s_{n-1} \rightarrow s_n$ and $\sigma \in \Sigma$ and there is some instantiation $\sigma' = \forall V'_1, V'_2, \ldots, V'_p.s'_1 \rightarrow \cdots \rightarrow s'_{n-1} \rightarrow t_m$, then following derivation is valid.

$$\frac{(\Sigma \vDash t_1 \rightarrow \cdots \rightarrow t_{m-1} \rightarrow s_i)^{i \in \{1,\ldots,n-1\}}}{\Sigma \vDash t_1 \rightarrow \cdots \rightarrow t_{m-1} \rightarrow t_m}\ refine$$

2.1 Defining a simple logic of programs

To illustrate logical development in the framework, let's define a simple logic of programming. We base this logic on a Martin–Löf-style type theory, which contains *programs*, and *types* that classify (and specify) the programs. The programs are defined inductively. We will include several simple programs including numbers, tuples, disjoint unions (variant records), functions, and function application, shown in Table 2.1.

Table 2.1: Syntax for a simple type theory

p	::=	$0 \mid s(0) \mid s(s(0)) \mid \ldots$	(numbers)
	|	**if** $p_1 = 0$ **then** p_2 **else** $v.p_3[v]$	
	|	$inl(p) \mid inr(p)$	(unions)
	|	**match** p_1 **with** $inl(x) \rightarrow p_2[x] \mid inr(y) \rightarrow p_3[y]$	
	|	$(p_1, p_2) \mid$ **match** p_1 **with** $(x, y) \rightarrow p_2[x; y]$	(pairs)
	|	$\lambda(v.p[v]) \mid p_1\ p_2$	(functions)

Table 2.2: Operational rules for the simple type theory

if $0 = 0$ **then** p_2 **else** $v.p_3[v]$	$\longrightarrow$	p_2
if $s(p_1) = 0$ **then** p_2 **else** $v.p_3[v]$.	$\longrightarrow$	$p_3[p_1]$
match $inl(p_1)$ **with** $inl(x) \rightarrow p_2[x] \mid inr(y) \rightarrow p_3[y]$	$\longrightarrow$	$p_2[p_1]$
match $inr(p_1)$ **with** $inl(x) \rightarrow p_2[x] \mid inr(y) \rightarrow p_3[y]$	$\longrightarrow$	$p_3[p_1]$
match (p_1, p_2) **with** $(x, y) \rightarrow p_3[x; y]$	$\longrightarrow$	$p_3[p_1; p_2]$
$\lambda(v.p_1[v])\ p_2$	$\longrightarrow$	$p_1[p_2]$

We can give an interpretation of these programs operationally by giving rules that define reduction on non-canonical programs (a canonical program is one for which no reduction rules apply). The operational rules are shown in Table 2.2. In these rules, and in the syntax definition, the expression $p[v]$ is meta-notation. On the left side of a rule, $p_1[v]$ represents a program with a possibly free occurrence of the variable v. The corresponding form $p_1[p_2]$ on the right side denotes *substitution* of the program p_2 for the variable v in the program p_1. It is important to be clear here: the bracketed forms are *not* part of the programming language; they are part of the description of its operation.

We will encode these programs in the meta-logic using a higher-order abstract syntax interpretation: variables in the programming language are interpreted as variables in the meta-logic. The meta-programming notation for substitution becomes second-order *application* in the meta-logic. First, we must give an interpretation of the syntax of programs as *terms*. The interpretation is shown in Table 2.3, using "semantic brackets" $[\![\cdot]\!]$ to formalize the interpretation.

In essence, for each of the program we choose a term in the meta-logic for its interpretation. The next step is to interpret the operational rules in the meta-logic, and for this we will rely on the second-order meta-variables to give the interpretation of the operational rules shown

Table 2.3: Interpretation of the syntax of programs in the simple type theory

program	term interpretation
$[\![v]\!]$	v
$[\![0]\!]$	$\mathtt{zero}$
$[\![s(i)]\!]$	$\mathtt{succ}([\![i]\!])$
$[\![\textbf{if } p_1 = 0 \textbf{ then } p_2 \textbf{ else } v.p_3[v]]\!]$	$\mathtt{ifzero}([\![p_1]\!];[\![p_2]\!];v.[\![p_3[v]]\!])$
$[\![inl(p)]\!]$	$\mathtt{inl}([\![p]\!])$
$[\![inr(p)]\!]$	$\mathtt{inr}([\![p]\!])$
$[\!\![\ \textbf{match } p_1 \textbf{ with } inl(x) \to p_2[x] \mid inr(y) \to p_3[y]\ ]\!\!]$	$\mathtt{decide}([\![p_1]\!];x.[\![p_2[x]]\!];y.[\![p_3[y]]\!])$
$[\![(p_1,p_2)]\!]$	$\mathtt{pair}([\![p_1]\!];[\![p_2]\!])$
$[\!\![\ \textbf{match } p_1 \textbf{ with } (x,y) \to p_2[x;y]\ ]\!\!]$	$\mathtt{spread}([\![p_1]\!];x,y.[\![p_2[x;y]]\!])$
$[\![\lambda(v.p[v])]\!]$	$\mathtt{lambda}(v.[\![p[v]]\!])$
$[\![p_1\ p_2]\!]$	$\mathtt{apply}([\![p_1]\!];[\![p_2]\!])$

in Table 2.4 (remember that $V_1, V_2, \ldots$ are second-order variables, and $V[t_1;\ldots;t_n]$ is a second-order application). There are a few things to note. First, since reduction is valid in any context, each reduction operation is interpreted consistently with an "outer" second-order application V_1. Second, the reduction notation *redex* $\longrightarrow$ *contractum* becomes a meta-implication $[\![redex]\!] \longrightarrow [\![contractum]\!]$ meaning that the *proof* of a statement containing the redex is also a proof of that statement when the redex is replaced by the contractum.

Table 2.4: Interpretation of the operational rules in the simple type theory

if_zero:	$V_1[\mathtt{ifzero}(\mathtt{zero};V_2;v.V_3[v].])$	$\longrightarrow$	$V_1[V_2]$
if_nzero:	$V_1[\mathtt{ifzero}(\mathtt{succ}(V_2);V_3;v.V_4[v].])$	$\longrightarrow$	$V_1[V_4[V_2]]$
dec_left:	$V_1[\mathtt{decide}(\mathtt{inl}(V_2);x.V_3[x];y.V_4[y])]$	$\longrightarrow$	$V_1[V_3[V_2]]$
dec_right:	$V_1[\mathtt{decide}(\mathtt{inr}(V_2);x.V_3[x];y.V_4[y])]$	$\longrightarrow$	$V_1[V_4[V_2]]$
spread:	$V_1[\mathtt{spread}(\mathtt{pair}(V_2;V_3);x,y.V_4[x;y])]$	$\longrightarrow$	$V_1[V_4[V_2;V_3]]$
beta:	$V_1[\mathtt{apply}(\mathtt{lambda}(v.V_2[v]);V_3)]$	$\longrightarrow$	$V_1[V_2[V_3]]$

2.2 Defining additional programming operations

Now that we have formalized the operational semantics, let's use the meta-logic to show that we can add recursion to the language without requiring additional programming primitives. Recursion (looping) is a basic programming tool, and we can add it by defining a fixpoint operation $\mu(v.p[v])$ in terms of the Y-combinator.

$$\begin{aligned} Y &\equiv \lambda(f.\lambda(x.f\ (x\ x))\ \lambda(x.f\ (x\ x))) \\ \mu(v.p[v]) &\equiv Y\ \lambda(v.p[v]). \end{aligned}$$

These definitions are standard, and we can add the *definitions* to the type theory as operational rules *unfold_y*: $V_1[Y] \longrightarrow V_1[[\![Y]\!]]$ and *unfold_fix*: $\mathtt{fix}(v.V_2[v]) \longrightarrow V_1[Y\ \lambda(v.V_2[v])]$. This characterization of the fixpoint is rather difficult to use; it will be convenient to *derive* a more-standard definition. Let Σ be the set of operational rules defined so far and let's assume that we also added their inverses (for instance "fold_beta" would be defined as $V_1[V_2[p_2]] \longrightarrow V_1[\mathtt{apply}(\mathtt{lambda}(v.V_1[v]); V_2)]$). We will want to derive the following judgment:

$$\Sigma \vDash V_1[\mathtt{fix}(v.V_2[v])] \longrightarrow V_1[V_2[\mathtt{fix}(v.V_2[v])]].$$

A proof is a sequence of steps, where each step applies a primitive meta-logical rule. We present the proof in an equational-reasoning style, where each step states the goal and a justification that is used to "refine" the goal. For brevity, we will mix the programming notation with the concrete meta-syntax; a sketch of the proof is shown in Table 2.5. Note that the final steps require the folding rules for reduction.

2.3 Defining a type theory for the simple programs

We will postpone discussion of proof automation for the moment, and move on to the development of a type system for the programming language. Types are a mechanism for characterizing classes of programs, and they are a means for adding *specifications* to the programming language. In a standard type system, there is a type classifier for each kind of program, and the primary logical judgment is a judgment of membership of a program in a type. We'll express type judgment in a sequent calculus. The sequents have the form

$$v_1 : T_1; v_2 : T_2[v_1]; \cdots; T_n[v_1; \ldots; v_{n-1}] \vdash p \in T_{n+1}[v_1; \ldots; v_n],$$

Table 2.5: Deriving the standard fixpoint reduction

$\Sigma, V_1[V_2[\mu(v.V_2[v])]] \vDash V_1[\mu(v.V_2[v])]$
BY unfold_fix
$\Sigma, V_1[V_2[\mu(v.V_2[v])]] \vDash V_1[\mathcal{Y}\ \lambda(v.V_2[v])]$
BY unfold_y (with $V_1[x] = V_1[x\ \lambda(v.V_2[v])]$)
$\Sigma, V_1[V_2[\mu(v.V_2[v])]] \vDash$
$V_1[\lambda(f.\lambda(x.f\ (x\ x))\ \lambda(x.f\ (x\ x)))\ \lambda(v.V_2[v])]$
BY beta
$\Sigma, V_1[V_2[\mu(v.V_2[v])]] \vDash$
$V_1[(\lambda(v.V_2[v])\ (x\ x))\ (\lambda(v.V_2[v])\ (x\ x))]$
BY beta
$\Sigma, V_1[V_2[\mu(v.V_2[v])]] \vDash$
$V_1[\lambda(v.V_2[v])\ ((\lambda(v.V_2[v])\ (x\ x))\ (\lambda(v.V_2[v])\ (x\ x)))]$
BY beta
$\Sigma, V_1[V_2[\mu(v.V_2[v])]] \vDash$
$V_1[V_2[((\lambda(v.V_2[v])\ (x\ x))\ (\lambda(v.V_2[v])\ (x\ x)))]]$
BY fold_beta
$\Sigma, V_1[V_2[\mu(v.V_2[v])]] \vDash$
$V_1[V_2[\lambda(f.\lambda(x.f\ (x\ x))\ \lambda(x.f\ (x\ x)))\ \lambda(v.V_2[v])]]$
BY fold_y
$\Sigma, V_1[V_2[\mu(v.V_2[v])]] \vDash V_1[V_2[\mathcal{Y}\ \lambda(v.V_2[v])]]$
BY fold_fix
$\Sigma, V_1[V_2[\mu(v.V_2[v])]] \vDash V_1[V_2[\mu(v.V_2[v])]]$
BY axiom

which expresses a hypothetical judgment. If v_1 is a program with type T_1, v_2 is a program with type $T_2[v_1]$, ..., and $T_n[v_1;\ldots;v_{n-1}$, then p is a program with type $T_{n+1}[v_1;\ldots;v_n]$.

Since we intend to use the types as specifications, we will use a *dependent* typing system, where the types of the programs can *depend* on the programs themselves. For example, what is the type of a the following program?

$$\texttt{decr} \equiv \lambda(i.\,\textbf{if}\; i = 0\; \textbf{then}\; 0\; \textbf{else}\; v.(i, v))$$

If the argument to the function is zero, it returns a number; otherwise it returns a pair of numbers. This is a somewhat contrived program, but is is certainly sensible. In our logic of specification, we would like to be able to describe this function, and state explicitly what its return values are. Let's assume that the natural numbers are described by the type $\mathbb{N}$, and that function spaces are denoted by the types $v{:}A \to B[v]$, where A is a type describing the domain of the function, and $B[v]$ is a program that computes the type of the result depending on the actual argument v. The `decr` function would have the following type:

$$\texttt{decr} \in i{:}\mathbb{N} \to (\textbf{if}\; i = 0\; \textbf{then}\; \mathbb{N}\; \textbf{else}\; \mathbb{N} \times \mathbb{N}.),$$

corresponding directly to our informal description.

Dependent types play a significant role in the specification of *partial* functions that are ill-formed either due to non-termination, or because they are not considered sensible (the program that adds a number and a pair might be considered nonsense). Arithmetic abounds in such examples. The fixpoint operation defined in the last section allows the definition of the standard arithmetic operators; a few examples are shown in Table 2.6.

We would expect that the standard operators $+$, $-$, would all return numbers if their arguments are numbers. The remainder is a special case, because it does not terminate if the divisor is zero. Can we express this in the type system? The answer is yes, but to do so, we will need to add a "void" type that contains *no* programs. The remainder would have two types, depending on the value of the second argument.

$$\begin{aligned}&\lambda(v_2.\lambda(v_1.v_1\ \textbf{rem}\ v_2))\\ &\quad \in v_2{:}\mathbb{N} \to \textbf{if}\; v_2 = 0\; \textbf{then}\; (\mathit{Void} \to \mathit{Void})\; \textbf{else}\; v_3.(\mathbb{N} \to \mathbb{N})\end{aligned}$$

The typing essentially states that if the divisor is zero, there is no program that will produce a result from the remainder.

So far, we have been informal in the discussion of types. Our first task in defining the type system will be deciding what *is* a type. The

Table 2.6: Some standard arithmetic operations in the simple type theory

$$
\begin{array}{rl}
p_1 + p_2 & \longrightarrow \ \mu(f.\lambda(v_1.\,\textbf{if}\, v_1 = 0\ \textbf{then}\ p_1\ \textbf{else}\ v_2.s(f\ v_2)))\ p_2 \\
p_1 - p_2 & \longrightarrow \\
\multicolumn{2}{l}{\mu(f.\lambda(v_1.\lambda(v_2.\ \textbf{if}\, v_2 = 0 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad))\ p_1\ p_2} \\
\multicolumn{2}{l}{\qquad\qquad\qquad\quad \textbf{then}\ v_1} \\
\multicolumn{2}{l}{\qquad\qquad\qquad\quad \textbf{else}\ v_3.\,\textbf{if}\, v_1 = 0\ \textbf{then}\ 0\ \textbf{else}\ v_4.f\ v_3\ v_4} \\
p_1 \le p_2 & \longrightarrow \ \textbf{if}\, p_1 - p_2 = 0\ \textbf{then}\ 1\ \textbf{else}\ v.0 \\
p_1\ \textbf{rem}\ p_2 & \longrightarrow \\
\multicolumn{2}{r}{\mu(f.\lambda(v_1.\,\textbf{if}\, v_1 \le p_2 = 0\ \textbf{then}\ f\ (v_1 - e_2)\ \textbf{else}\ v_2.v_1))\ p_1}
\end{array}
$$

Table 2.7: The syntax of types for the simple type theory

T	::=	p	(programs)
	|	*Void*	(empty type)
	|	$\mathbb{N}$	(numbers)
	|	$T_1 + T_2$	(unions)
	|	$v{:}T_1 \times T_2[v]$	(pairs)
	|	$v{:}T_1 \to T_2[v]$	(functions)

Table 2.8: Typehood judgments for the simple type theory

$$\frac{}{\Gamma \vdash \mathbb{N} \in \mathit{Type}}\ \mathit{nat} \qquad \frac{\Gamma \vdash T_1 \in \mathit{Type} \quad \Gamma \vdash T_2 \in \mathit{Type}}{\Gamma \vdash T_1 + T_2 \in \mathit{Type}}\ \mathit{union}$$

$$\frac{\Gamma \vdash T_1 \in \mathit{Type} \quad \Gamma; v{:}T_1 \vdash T_2[v] \in \mathit{Type}}{\Gamma \vdash v{:}T_1 \times T_2[v] \in \mathit{Type}}\ \mathit{prod}$$

$$\frac{\Gamma \vdash T_1 \in \mathit{Type} \quad \Gamma; v{:}T_1 \vdash T_2[v] \in \mathit{Type}}{\Gamma \vdash v{:}T_1 \to T_2[v] \in \mathit{Type}}\ \mathit{fun}$$

syntax of types includes the syntax of programs, shown in Table 2.7. Of course, by including the programs, some of these expressions will not be types (for example, `zero` is not a type). We will define the well-formed types inductively, using a sequent calculus to state proofs of typehood judgment. The rules for typehood start from the base type $\mathbb{N}$, and work upward through types for pairs and functions, shown in Table 2.8. The most interesting judgments are for the dependent types $v{:}T_1 \times T_2[v]$ and $v{:}T_1 \to T_2[v]$: in both cases, the expression $T_2[v]$ must be a type for *any* program $v \in T_1$.

These rules are described abstractly: the meta-variable Γ denotes any sequence of assumptions in the sequent. For the interpretation of the rules in the meta-logic, we first need to give an interpretation of the types in the syntax of terms in the meta-logic. This translation, shown in Table 2.9 is an extension of the interpretation of programs (programs occurring in the types are interpreted by the definitions in Table 2.3). This table also lists the coding for *sequents* as a recursive definition.

The rules can be interpreted in the meta-logic in the same way as for the operational definitions. We interpret the sequent meta-variable Γ as a second-order variable that defines the sequent *context*. For example, the rule *fun* has the following interpretation, where the second-order variable represents the context Γ. Technically, we also have a well-formedness restriction in this interpretation: any refinement of this implication made by substituting for V_1 must be a sequent. This can be formalized as a type judgment, and for simplicity we omit analysis of sequent well-formedness in this introduction.

$$\begin{aligned} \textit{fun}\text{:}\ & V_1[\mathtt{concl}(\mathtt{type}(V_1))] \longrightarrow \\ & V_1[\mathtt{hyp}(V_1; v.\mathtt{concl}(V_2[v]))] \longrightarrow \\ & V_1[\mathtt{concl}(\mathtt{type}(\mathtt{dfun}(V_1; v.V_2[v])))] \end{aligned}$$

Before moving to the issue of proofs, let's also define the type membership judgments in the type theory: when does a program have a type? The type judgment is expressed as a membership $p \in T$, and

Table 2.9: Interpretation of types in the simple type theory

$$
\begin{array}{rcl}
[\![\mathit{Void}]\!] & \longrightarrow & \texttt{void} \\
[\![\mathbb{N}]\!] & \longrightarrow & \texttt{nat} \\
[\![T_1 + T_2]\!] & \longrightarrow & \texttt{union}([\![T_1]\!]; [\![T_2]\!]) \\
[\![v{:}T_1 \times T_2[v]]\!] & \longrightarrow & \texttt{dprod}([\![T_1]\!]; v.[\![T_2[v]]\!]) \\
[\![v{:}T_1 \to T_2[v]]\!] & \longrightarrow & \texttt{dfun}([\![T_1]\!]; v.[\![T_2[v]]\!]) \\
\\
[\![T[v_1;\ldots;v_n] \in \mathit{Type}]\!] & \longrightarrow & \texttt{type}([\![T[v_1;\ldots;v_n]]\!]) \\
[\![\vdash \sigma]\!] & \longrightarrow & \texttt{concl}([\![\sigma]\!])
\end{array}
$$

$$
\begin{array}{lll}
\text{Let } \sigma' = & v_{i+1}: & T_{i+1}[v_1;\ldots;v_i]; \\
& \cdots & \\
& v_n: & T_n[v_1;\ldots;v_{[}n-1]] \\
& \vdash & \sigma \\
[\![v_i{:}T_i[v_1;\ldots;v_{i-1}];\sigma']\!] & \longrightarrow & \texttt{hyp}([\![T_i[v_1;\ldots;v_{i-1}]]\!]; v_i.[\![\sigma']\!])
\end{array}
$$

there are two kinds of rules: if we view the proof as a derivation from first principles, *introduction* rules are used to derive membership judgments of programs from their parts, and *elimination* rules are used to build larger assumptions. In a backward-chaining style of reasoning, introduction rules decompose type judgments on the right of the sequent turn-stile, and elimination rules decompose types on the left. The introduction rules are shown in Table 2.10, and the elimination rules are shown in Table 2.11.

The tables mix the notation of the object and meta-languages, but they can be translated directly into the meta-logic. The Γ and $\Delta[p]$ correspond to second-order contexts (they represents arbitrary sequences of hypotheses), and the variables p, p_1, p_2, T_1, T_2 are also second-order.

There are a few things to note about these rules. The elimination forms correspond to *induction* principles (witness the `int` elimination rule). The type dependencies make the rules fairly complicated, but in many cases these dependencies are only needed for specifications (as we will see in the next section).

Proofs in this type theory follow the same process as the reduction proofs (in Table 2.5 for example), but the sequence of steps in general forms a proof tree rather than a linear sequence of reasoning steps. To illustrate the proof process, let's show that the addition function we

Table 2.10: Membership introduction rules in the simple type theory

$$\frac{}{\Gamma \vdash 0 \in \mathbb{N}}\ zero \qquad \frac{\Gamma \vdash i \in \mathbb{N}}{\Gamma \vdash s(i) \in \mathbb{N}}\ succ$$

$$\frac{\Gamma \vdash p_1 \in \mathbb{N} \quad \Gamma \vdash p_2 \in T[0] \quad \Gamma,\ i:\mathbb{N} \vdash p_3[i] \in T[i]}{\Gamma \vdash \textbf{if}\ p_1 = 0\ \textbf{then}\ p_2\ \textbf{else}\ v.p_3[v] \in T[p_1]}\ ifzero$$

$$\frac{\Gamma \vdash p \in T_1 \quad \Gamma \vdash T_2 \in Type}{\Gamma \vdash inl(p) \in T_1 + T_2}\ inl \qquad \frac{\Gamma \vdash p \in T_2 \quad \Gamma \vdash T_1 \in Type}{\Gamma \vdash inr(p) \in T_1 + T_2}\ inr$$

$$\frac{\Gamma \vdash p_1 \in T_1 \quad \Gamma \vdash p_2 \in T_2[p_1] \quad \Gamma,\ v{:}T_1 \vdash T_2[v] \in Type}{\Gamma \vdash (p_1, p_2) \in (v{:}T_1) \times T_2[v]}\ pair$$

$$\frac{\Gamma \vdash p_1 \in (v{:}T_1) \times T_2[v] \quad \Gamma,\ v_1{:}T_1,\ v_2{:}T_2[v_1] \vdash p_2[v_1,\ v_2] \in T[(v_1, v_2)]}{\Gamma \vdash \textbf{match}\ p_1\ \textbf{with}\ (v_1, v_2) \to p_2[v_1,\ v_2] \in T[p_1]}\ spread$$

$$\frac{\Gamma,\ v_1{:}T_1 \vdash p[v_1] \in T_2[v_1] \quad \Gamma \vdash T_1 \in Type}{\Gamma \vdash \lambda(v_1.p[v_1]) \in (v_2{:}T_1) \to T_2[v_2]}\ lambda$$

$$\frac{\Gamma \vdash p_1 \in (v_1{:}T_1) \to T_2[v_2] \quad \Gamma,\ v_1{:}T_1 \vdash p_2 \in T_2[v_1]}{\Gamma \vdash p_1\ p_2 \in T_2[p_1]}\ apply$$

Table 2.11: Structural and elimination rules in the simple type theory

$$\frac{}{\Gamma,\ v{:}T,\ \Delta[t] \vdash x \in T}\ axiom \qquad \frac{\Gamma \vdash T_2 \in p \quad \Gamma,\ v{:}T_2 \vdash T_1[v]}{\Gamma \vdash T_1[p]}\ gen$$

$$\frac{\Gamma,\ i{:}\mathbb{N},\ \Delta[i] \vdash T[0] \quad \Gamma,\ i{:}\mathbb{N},\ \Delta[i],\ j{:}\mathbb{N},\ T[j] \vdash T[s(j)]}{\Gamma,\ i{:}\mathbb{N},\ \Delta[i] \vdash T[i]}\ nat$$

$$\frac{\Gamma,\ p{:}T_1,\ \Delta[inl(p)] \vdash T[inl(p)] \quad \Gamma,\ p{:}T_2,\ \Delta[inr(p)] \vdash T[inr(p)]}{\Gamma,\ p{:}T_1 + T_2,\ \Delta[p] \vdash T[p]}\ union$$

$$\frac{\Gamma,\ p_1{:}T_1,\ p_2{:}T_2[p_1],\ \Delta[(p_1, p_2)] \vdash T[(p_1, p_2)]}{\Gamma,\ p{:}(v_1{:}T_1) \times T_2[v_1],\ \Delta[p] \vdash T[p]}\ pair$$

$$\frac{\begin{array}{l}\Gamma,\ p{:}(v_1{:}T_1) \to T_2[v_1],\ \Delta[p] \vdash a \in T_1 \\ \Gamma,\ p{:}(v_1{:}T_1) \to T_2[v_1],\ \Delta[p],\ p_2{:}T_2[p_1] \vdash T[p]\end{array}}{\Gamma,\ p{:}(v_1{:}T_1) \to T_2[v_1],\ \Delta[p] \vdash T[p]}\ fun$$

Table 2.12: Typing of addition in the simple type theory

$\Sigma_1, \Sigma_2 \vDash (\Gamma \vdash p_1 + p_2 \in \mathbb{N})$
BY gen over $p_2 \in \mathbb{N}$
$\Sigma_1, \Sigma_2 \vDash (\Gamma,\ v{:}\mathbb{N} \vdash p_1 + v \in \mathbb{N})$
BY nat elimination
-1. $\Sigma_1, \Sigma_2 \vDash (\Gamma \vdash p_1 + 0 \in \mathbb{N})$
BY *reduction for addition*
$\Sigma_1, \Sigma_2 \vDash (\Gamma \vdash p_1 \in \mathbb{N})$
BY assumption in Σ_2
-2. $\Sigma_1, \Sigma_2 \vDash (\Gamma,\ v{:}\mathbb{N},\ p_1 + v \in \mathbb{N} \vdash p_1 + s(v) \in \mathbb{N})$
BY *reduction for addition*
$\Sigma_1, \Sigma_2 \vDash (\Gamma,\ v{:}\mathbb{N},\ p_1 + v \in \mathbb{N} \vdash s(p_1 + v) \in \mathbb{N})$
BY add introduction
$\Sigma_1, \Sigma_2 \vDash (\Gamma,\ v{:}\mathbb{N},\ p_1 + v \in \mathbb{N} \vdash p_1 + v \in \mathbb{N})$
BY axiom

defined has a type. The desired typing judgment is this:

$$\frac{\Gamma \vdash p_1 \in \mathbb{N} \quad \Gamma \vdash p_2 \in \mathbb{N}}{\Gamma \vdash p_1 + p_2 \in \mathbb{N}}\ add.$$

We can derive the rule in a sequence of steps, applying the primitive rules just defined. The proof is shown in Table 2.12, where Σ_1 are all the rules we have defined so far, and Σ_2 are the assumptions $\Gamma \vdash p_1 \in \mathbb{N}$ and $\Gamma \vdash p_2 \in \mathbb{N}$.

For brevity, the reductions are omitted. The important step in the proof is the first one: a generalization of the second argument of the addition function that allows the use of induction in the proof. The choice of the expression to generalize depends on the structure of the addition function. In this case, expression p_2 is generalized because we have the foresight to know that we can prove $p_1 + s(p_2) \longrightarrow s(p_1 + p_2)$. Steps like these will be the major obstacle to proof automation.

2.4 Defining a logic

So far we have a programming language and a system of types. The types are quite expressive, but they are not the most natural way to express specifications. For example, we can prove $\lambda(x.\lambda(y.x+y)) \in \mathbb{N} \to \mathbb{N} \to \mathbb{N}$, but it would be much more natural to define a logic of quantification that would allow us to say more directly that "the addition

Table 2.13: Interpretation of propositions as type

$$
\begin{array}{rcl}
\texttt{false} & \longrightarrow & \texttt{void} \\
\texttt{and}(A;B) & \longrightarrow & \texttt{dprod}(A;v.B) \\
\texttt{or}(A;B) & \longrightarrow & \texttt{union}(A;B) \\
\texttt{implies}(A;B) & \longrightarrow & \texttt{dfun}(A;v.B) \\
\texttt{exists}(A;v.B[v]) & \longrightarrow & \texttt{dprod}(A;v.B[v]) \\
\texttt{all}(A;v.B[v]) & \longrightarrow & \texttt{dfun}(A;v.B[v])
\end{array}
$$

function returns a number if its arguments are numbers." To do this, we will extend the type theory with a constructive first-order predicate calculus. The syntax of predicates is defined inductively:

$\top$ (true) and $\bot$ (false) are predicates.

If A and B are predicates, so are:
$A \wedge B$ (conjunction)
$A \vee B$ (disjunction)
$A \Rightarrow B$ (implication)
If T is a type, and $A[v]$ is a predicate with a free variable v, then
$\forall v{:}T.A[v]$ (universal quantification)
$\exists v{:}T.A[v]$ (existential quantification)
If T is a type, and p is a program, then
$p \in T$ is a predicate.

The quantifiers $\forall v{:}T.a[v]$ and $\exists v{:}T.A[v]$ quantify over all *programs* having type T. We will want to mix predicates and programs. As usual, the translation of the syntax assigns a term $\texttt{true}, \texttt{false}, \texttt{and}\{a;b\}, \ldots$ to the predicates $\top, \bot, A \wedge B, \ldots$.

The most natural interpretation of the predicates uses the propositions-as-types interpretation where *programs* stand for *proofs*, and *propositions* (predicates) stand for *types*. The interpretation of the propositions can be defined using rewrite definitions in the meta-logic, shown in Table 2.13. These rewrites are *definitions* that translate the new syntax of propositions to the existing syntax of types.

The logic of propositions is a logic of proofs, where a proposition is *true* if there is a program that has the proposition as its type. For instance, the proposition $\bot \Rightarrow \bot$ is true because the program $\lambda(x.x)$ has the type $(v{:}\mathit{Void}) \rightarrow \mathit{Void}$. The rules of the logic are stated in introduction/elimination form, shown in Table 2.14.

Since the propositions are defined as types, we should be able to *derive* the logical rules from the membership rules of the type theory.

Table 2.14: Predicate rules in the simple type theory

$$\frac{\Gamma \vdash a \in A}{\Gamma \vdash A}\ \textit{witness} \qquad \frac{\Gamma \vdash A \quad \Gamma,\ v{:}A \vdash B}{\Gamma \vdash B}\ \textit{cut}$$

$$\frac{}{\Gamma \vdash \top}\ \textit{true-intro} \qquad \frac{}{\Gamma,\ v{:}\bot,\ \Delta[v] \vdash C[v]}\ \textit{false-elim}$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \textit{and-intro}$$

$$\frac{\Gamma,\ v_1{:}A,\ v_2{:}B,\ \Delta[(v_1, v_2)] \vdash C[(v_1, v_2)]}{\Gamma,\ v{:}A \wedge B,\ \Delta[v] \vdash C[v]}\ \textit{and-elim}$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B \in \mathit{Type}}{\Gamma \vdash A + B}\ \textit{or-left-intro}$$

$$\frac{\Gamma \vdash B \quad \Gamma \vdash A \in \mathit{Type}}{\Gamma \vdash A + B}\ \textit{or-right-intro}$$

$$\frac{\begin{array}{c}\Gamma,\ v{:}A,\ \Delta[\mathit{inl}(v)] \vdash C[\mathit{inl}(v)] \\ \Gamma,\ v{:}B,\ \Delta[\mathit{inr}(v)] \vdash C[\mathit{inr}(v)]\end{array}}{\Gamma,\ v{:}A + B,\ \Delta[v] \vdash C[v]}\ \textit{or-elim}$$

$$\frac{\Gamma \vdash A \in \mathit{Type} \quad \Gamma,\ v{:}A \vdash B}{\Gamma \vdash A \Rightarrow B}\ \textit{imp-intro}$$

$$\frac{\begin{array}{c}\Gamma,\ v{:}A \Rightarrow B,\ \Delta[v] \vdash A \\ \Gamma,\ v{:}A \Rightarrow B,\ \Delta[v],\ v_2{:}B \vdash C[v]\end{array}}{\Gamma,\ v{:}A \Rightarrow B,\ \Delta[v] \vdash C[v]}\ \textit{imp-elim}$$

$$\frac{\Gamma \vdash p \in T \quad \Gamma \vdash A[p]}{\Gamma \vdash \exists v{:}T.A[v]}\ \textit{exists-intro}$$

$$\frac{\Gamma,\ v_1{:}T,\ v_2{:}A[v],\ \Delta[(v_1, v_2)] \vdash C[(v_1, v_2)]}{\Gamma,\ v{:}\exists v{:}T.A[v],\ \Delta[v] \vdash C[v]}\ \textit{exists-elim}$$

$$\frac{\Gamma \vdash T \in \mathit{Type} \quad \Gamma,\ v{:}T \vdash A[v]}{\Gamma \vdash \forall v{:}T.A[v]}\ \textit{all-intro}$$

$$\frac{\begin{array}{c}\Gamma,\ v{:}\forall v{:}T.A[v],\ \Delta[v] \vdash p \in T \\ \Gamma,\ v{:}\forall v{:}T.A[v],\ \Delta[v],\ v_2{:}A[p] \vdash C[v]\end{array}}{\Gamma,\ v{:}\forall v{:}T.A[v],\ \Delta[v] \vdash C[v]}\ \textit{all-elim}$$

Table 2.15: Derivation of and-introduction

$\Sigma \vDash (\Gamma \vdash A) \longrightarrow (\Gamma \vdash B) \longrightarrow (\Gamma \vdash A \wedge B)$
BY cut A
- $\Sigma \vDash (\Gamma \vdash A) \longrightarrow (\Gamma \vdash B) \longrightarrow (\Gamma \vdash A)$
  BY assumption
- $\Sigma \vDash (\Gamma \vdash A) \longrightarrow (\Gamma \vdash B) \longrightarrow (\Gamma,\ v{:}A \vdash A \wedge B)$
  BY cut B
  - $\Sigma \vDash (\Gamma \vdash A) \longrightarrow (\Gamma \vdash B) \longrightarrow (\Gamma,\ v{:}A \vdash B)$
    BY thin A
    $\Sigma \vDash (\Gamma \vdash A) \longrightarrow (\Gamma \vdash B) \longrightarrow (\Gamma \vdash B)$
    BY assumption
  - $\Sigma \vDash (\Gamma \vdash A) \longrightarrow (\Gamma \vdash B) \longrightarrow (\Gamma,\ v_1{:}A,\ v_2{:}B \vdash A \wedge B)$
    BY witness (v_1, v_2)
    $\Sigma \vDash (\Gamma \vdash A) \longrightarrow (\Gamma \vdash B) \longrightarrow$
    $(\Gamma,\ v_1{:}A,\ v_2{:}B \vdash (v_1, v_2) \in A \wedge B)$
    BY unfold_and
    $\Sigma \vDash (\Gamma \vdash A) \longrightarrow (\Gamma \vdash B) \longrightarrow$
    $(\Gamma,\ v_1{:}A,\ v_2{:}B \vdash (v_1, v_2) \in (v_3{:}A) \times B)$
    BY pair
    - $\Sigma \vDash (\Gamma \vdash A) \longrightarrow (\Gamma \vdash B) \longrightarrow (\Gamma,\ v_1{:}A,\ v_2{:}B \vdash v_1 \in A)$
      BY axiom
    - $\Sigma \vDash (\Gamma \vdash A) \longrightarrow (\Gamma \vdash B) \longrightarrow (\Gamma,\ v_1{:}A,\ v_2{:}B \vdash v_2 \in B)$
      BY axiom

As it turns out, we can derive most, but not all of the rules. An example derivation of the rule for and-intro is shown in Table 2.15.

Most of the other rules have similar proofs, but the dependent predicate introduction rules are not directly derivable. A derivation of *all-intro* requires a proof for:

$$(\Gamma,\ v{:}A \vdash B[v]) \longrightarrow (\Gamma \vdash (v{:}A) \to B[v]).$$

If we had a witness $b[v] \in B[v]$, then a witness for the goal would be $\lambda(v.b[v])$. We know that such a witness exists: it can be constructed by induction on proofs. In fact, the construction of such witnesses is a cornerstone of constructive reasoning, and the MetaPRL implementation constructs these witnesses during proof validation.

There are a few things to point out before moving to the next section. First, here are the steps we took in designing a programming language and logic:

1. defined the syntax of program primitives using terms,
2. defined the operational semantics (semantics of reduction) as meta-logical rewrite definitions,
3. derived some operational properties of some non-primitive programs,
4. defined the types of programs using terms,
5. encoded the membership judgment as rules in the meta-logic (using second-order variables to interpret the rule meta-variables),
6. defined a logic of programs by giving a translation between logics and types, and deriving most of the logical rules from the rules for type membership.

This was a fairly long process, but it is still not complete. For one, the proofs in the logic are quite tedious: they require manual application of the three rules of the meta-logic. For larger programs, the size of the proof could easily grow to require many thousands of steps. Furthermore, much of the reasoning is quite standardized: if we're trying to prove $A \wedge B$, it is almost always a good idea to prove it from two subproofs: a proof of A and a proof of B. This brings us to the issue of proof automation, and the MetaPRL system.

2.5 Proof automation

One of our goals with MetaPRL is to provide the ability not only to formalize logics, but also to help automate the reasoning about formally defined systems. The implementation is quite extensive (as we describe in Part III). It includes a theorem prover, a compiler, a manager for libraries of formal information, a proof editor for interactive construction of formal proofs, and several logics (described in Part V). For this introduction, we briefly describe the logical part of the MetaPRL system that we use in the remainder of the introduction.

MetaPRL is a foundational LCF-style theorem prover. Like other LCF-style tactic provers, such as Coq [DFH+93] and Nuprl [CAB+86], proofs are performed in a backward-chaining goal-directed style. Given a *goal* sentence like the sequent $A, A \Rightarrow B \vdash B$, the user selects a *tactic* to apply to the sentence in an attempt either to prove it, or reduce it to a set of simpler *subgoals*. When a tactic is applied, it reduces the proof to the set of primitive inferences defined by the logic. For example, a proof of the sequent above is given in Figure 2.1. The dT tactic is a generic tactic for

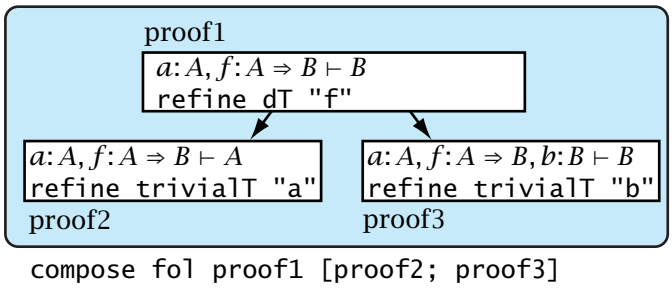

Figure 2.1: Example propositional proof

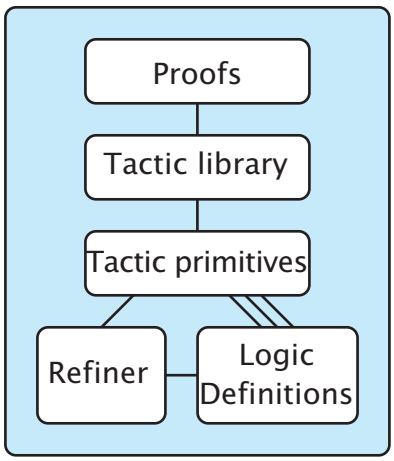

Figure 2.2: Tactic-prover architecture

applying introduction and elimination rules, and the `trivialT` tactic applies "trivial" rules like the AXIOM rule of propositional logic.

The proving architecture is implemented in several parts, shown in Figure 2.2. For each logic, there is a module defining the primitive rules of the logic. There is a logic engine, called the *refiner*, for applying a primitive rule to a goal to reduce it to a (possibly empty) set of subgoals, and there is a *primitive tactic* module that defines the basic tactic operations. Proof automation is implemented in the *tactic library*.

For each primitive rule and rewrite definition in a logic, the refiner defines a primitive object that can be used to apply the rule. In addition, the refiner exports the operations in Table 2.16.

The `term` type is used to represent logical sentences. The `rule` and `logic` types are used to represent primitive rules and logics. The `proof` type represents *partial* proof trees that have been constructed by refine-

Table 2.16: Refiner operations

```
module Refiner :  sig
  type term, rule, logic, proof
  exception RefineError
  val refine :  logic → rule → term → proof
  val compose :  proof → proof list → proof
  val goal_of_proof :  proof → term
  val subgoals_of_proof :  proof → term list
end
```

Table 2.17: Tactic primitives

```
module TacticPrimitives (Refiner) :  sig
  type prim_tactic, proof
  type tactic = term → prim_tactic
  val tactic_of_rule :  Refiner.rule → tactic
  val proof_of_goal :  logic → term → proof
  val refine :  proof → tactic → proof
  val andthenT : tactic → tactic → tactic
  val orelseT : tactic → tactic → tactic
end
```

ment; the `refine` function applies a `rule` to a goal to produce a partial proof. The `compose` function is used to compose partial proof trees. It is an error if any of the subgoals of the proof are not the same as the goals of the subproofs. If an error occurs at any time, the `RefineError` exception is raised.

The tactic-primitives module simplifies the application of the refiner's rules by supporting proof search and composition with *tacticals*. The tactic interface is defined in Table 2.17.

The tactic-primitives module re-implements the `proof` type to contain single node proof trees. It contains a constructor for building tactics from rules, and it implements a `refine` function that works with tactics. The `tactic` type itself is implemented as a function that returns a primitive reasoning operation, given the goal the tactic is being applied to. The tactic module also implements two *tacticals:* the `(andthenT tac1 tac2)` tactic applies `tac1` to the goal, then applies `tac2` to each of the subgoals. The `(orelseT tac1 tac2)` tactic applies `tac1` to the goal. If it succeeds, the proof is returned; otherwise,

Table 2.18: Proof procedure for propositional logic

```
(* Refiner declarations:
  * val imp_elim :  int → tactic
  * val imp_intro :  tactic
  * val axiom :  int → tactic
  *)
let rec autoT t =
  orelseT (onSomeHypT 0 imp_elim)
    (orelseT (onSomeHypT 0 axiom) imp_intro)
and onSomeHypT tac i t =
  if i = hyp_count t then
    raise RefineError
  else
    orelseT (tac i thenT autoT)
      (onSomeHypT tac (i + 1)) t
```

`tac2` is applied instead.

When embedded in a meta-language like OCaml, these primitives are sufficient to encode proof search. For example, a tactic to prove propositional sentences with implication is shown in Figure 2.18. The `axiom`. `imp_elim`, and `imp_intro` tactics are constructed from the rules given by the refiner. The `onSomeHypT` tactical searches the hypothesis of the goal sequent for a successful application of the `imp_elim` or `axiom` rules.

The implementation of the tactic primitives is straightforward. The `prim_tactic` type is implemented as the refiner type `proof`, the `tactic_of_rule` function calls the refiner with the goal argument, and the `refine` function is just the identity. The `andthenT tac1 tac2` tactical performs the refinements specified by the tactics `tac1` and `tac2`, then composes the result. The `orelseT tac1 tac2` tactical applies `tac1`, catches the exception and applies `tac2` if `tac1` failed.

These definitions define a basis for proof automation. When a theorem is to be proved, the statement of the theorem becomes the *goal* of the proof. The primitive rules and rewrites defined in a formal system become primitive tactics that can be applied to simplify in an attempt to find a proof. Furthermore, the tactics can be composed with the `TacticPrimitives` to form complex proof search procedures. For the remainder of this introduction, we assume that proofs are performed in the MetaPRL system, and we present the example proofs as *tactic-trees*: trees where each node has a goal, and a tactic that is used to simplify

Table 2.19: Tactic implementations

```
module Tactics (Refiner) = struct
  type prim_tactic = logic → proof
  let tactic_of_rule rule t logic =
    Refiner.refine logic t
  let andthenT tac1 tac2 t logic =
    let p = tac1 t logic in
    let tl = Refiner.subgoals_of_proof p in
    let pl = List.map (fun t → tac2 t logic) tl in
      compose p pl
  let orelseT tac1 tac2 t logic =
    try tac1 t logic with
    RefineError → tac2 t logic
end
```

the goal.

Chapter 3

Reasoning about programs

This thesis focuses on formal support for building large systems. Our implementation of the proof automation framework just presented is a relatively large system itself; how can we best reason about it? To explore this issue, we explore the problem of building large systems with components that have side-effects. Ultimately, the task will involve reasoning about program code. The logical foundation is program *semantics*: a mathematical interpretation of program code.

There are many types of semantics, including denotational, axiomatic, operational, and others. The simplest of these is probably the operational semantics, which describes program execution. In this section, we will show how to build an operational semantics of a simple imperative programming language using the type theory of Chapter 2. In the process, we will get to the main issue of relating logical theories. We will also see how the semantics reveals fundamental properties of the program code that characterize requirements for composition. Composition is the key to reasoning about large systems.

This section proceeds from the bottom up, from the most technical to the most abstract, in these parts.

1. Defining an imperative language
2. Defining the operational semantics
3. Defining and deriving a faithful interpretation from the simple type theory
4. Describing requirements on composition that enable composing justifications of program properties.

Table 3.1: Syntax of simple imperative programs

e	::=	$0, 1, 2, \ldots$	(numbers)
	|	$e_1 + e_2 \mid e_1 - e_2 \mid \ldots$	
	|	**if** $e_1 = e_2$ **then** e_3 **else** e_4	
	|	$inl(e) \mid inr(e)$	(unions)
	|	**match** e_1 **with** $inl(x) \rightarrow e_2[x] \mid inr(y) \rightarrow e_3[y]$	
	|	$(e_1, e_2) \mid$ **match** e_1 **with** $(x, y) \rightarrow e_2[x; y]$	(pairs)
	|	**match** e_1 **with** $(x, y) \rightarrow e_2[x; y]$	
	|	**ref** $e \mid !e \mid e_1 \leftarrow e_2$	(reference cells)
	|	$\lambda(x.e[x]) \mid e_1\ e_2$	(functions)

Table 3.2: State operations for the simple imperative language

$l_1, l_2, l_3, \ldots$	(Locations)
$[\,]$	(empty state)
$alloc(s, v)$	(allocation)
$s.l$	(dereferencing)
$s[l \leftarrow v]$	(assignment)

3.1 Defining an imperative language

First, let's define the syntax of a simple language. We'll include the integers as a base type, disjoint unions (variant records), tuples, functions, assignments, and sequencing of program code. This is similar to the simple functional language defined in Chapter 2, but this language allows side-effects in any expression, whereas functional language in Chapter 2 allows no side-effects at all. The syntax of the language is shown in Table 3.1.

Imperative programs can be viewed as state transformers, and the interpretation of these programs requires that we model the program state. When a program is initialized, the values of its variables are undefined, and we denote the empty state with the symbol $[\,]$. There are three operations on the state, corresponding to allocation of new reference cell, dereferencing (fetching the contents of) a cell, and assignment of a new value to a cell. The program state provides the storage for reference cells, and we will call the names of these fields *locations*; locations correspond to *pointers* in most programming languages. We will also use the notation in Table 3.2 for the state operations.

It is important to note that locations need not be ordered, but we

Table 3.3: Operational description of state operations

$$
\begin{array}{rcll}
s\,[l_1 \leftarrow v\,].l_2 & \longrightarrow & v & (\text{if } l_1 = l_2) \\
s\,[l_1 \leftarrow v\,].l_2 & \longrightarrow & s.l_2 & (\text{if } l_1 \neq l_2) \\
\mathit{alloc}(s, v) & \longrightarrow & (l, s\,[l \leftarrow v\,]) & l \notin \mathbf{dom}(s) \\
\mathbf{dom}([\,]) & \longrightarrow & \{\} & \\
\mathbf{dom}(s\,[l \leftarrow v\,]) & \longrightarrow & \{l\} \cup \mathbf{dom}(s) &
\end{array}
$$

Table 3.4: Definition of values for simple imperative programs

$$\frac{}{\mathit{value}(i)}\ \mathit{num\text{-}val} \qquad \frac{\mathit{value}(v)}{\mathit{value}(\mathit{inl}(v))}\ \mathit{inl\text{-}val} \qquad \frac{\mathit{value}(v)}{\mathit{value}(\mathit{inr}(v))}\ \mathit{inr\text{-}val}$$

$$\frac{\mathit{value}(v_1) \quad \mathit{value}(v_2)}{\mathit{value}((v_1, v_2))}\ \mathit{pair\text{-}val} \qquad \frac{}{\mathit{value}(\lambda(x.e[x]))}\ \mathit{fun\text{-}val}$$

$$\frac{}{\mathit{value}(\&l)}\ \mathit{loc\text{-}val} \qquad \frac{}{\mathit{value}(\cdot)}\ \mathit{dot\text{-}val}$$

will assume that equality on labels is computable. The state itself can be viewed as a record, mapping locations to values. The operational description of the state operations is given in Table 3.3. In this definition, the $\mathit{alloc}(s, v)$ operation is *nondeterministic*. Allocation corresponds to assignment of a value to a *new* location in the state, and any choice of new location is valid, as long as it does not correspond to a previously assigned location. The allocation returns the new label, and the new state.

The operational semantics of the imperative programs views the expressions in the language as state transformers, where each expression takes the current state and produces a value *and* a new state. Values are defined inductively: numbers, inl, inr, pairs, and functions are values, and there are two additional values that are not part of the program syntax: a "pointer" value $\&l$ that represents location l in the state, and a value $\cdot$ that represents the value of an assignment. The value judgment is shown in Table 3.4.

We will use the notation $(e, s_1) \downarrow (v, s_2)$ to say that expression e in state s_1 evaluates to a *value* v and produces state s_2. Table 3.5 gives the operational semantics in terms of this evaluation relation. There are a few things to note. First, this is a natural-semantics-style description: it only describes those programs that compute to a value. Second, functions are considered values: no evaluation occurs in the body of

Table 3.5: Natural semantics for simple imperative programs

$$\frac{}{(i, s) \downarrow (i, s)}\ \mathit{num}$$

$$\frac{(e, s_1) \downarrow (v, s_2)}{(\mathit{inl}(e), s_1) \downarrow (\mathit{inl}(v), s_2)}\ \mathit{inl} \qquad \frac{(e, s_1) \downarrow (v, s_2)}{(\mathit{inr}(e), s_1) \downarrow (\mathit{inr}(v), s_2)}\ \mathit{inr}$$

$$\frac{(e_1, s_1) \downarrow (v_1, s_2) \quad (e_2, s_2) \downarrow (v_2, s_3)}{((e_1, e_2), s_1) \downarrow ((v_1, v_2), s_3)}\ \mathit{pair}$$

$$\frac{(e_1, s_1) \downarrow ((v_1, v_2), s_2) \quad (e_2[v_1; v_2], s_2) \downarrow (v_3, s_3)}{((\mathbf{match}\, e_1\ \mathbf{with}\ (x, y) \to e_2[x; y]), s_1) \downarrow (v_3, s_3)}\ \mathit{spread}$$

$$\frac{(e, s_1) \downarrow (v, s_2) \quad \mathit{alloc}(s_2, v) = (l, s_3)}{(\mathbf{ref}\, e, s_1) \downarrow (\&l, s_3)}\ \mathit{ref} \qquad \frac{(e, s_1) \downarrow (\&l, s_2)}{(!e, s_1) \downarrow (s_2.l, s_2)}\ \mathit{deref}$$

$$\frac{(e_1, s_1) \downarrow (\&l, s_2) \quad (e_2, s_2) \downarrow (v_2, s_3)}{(e_1 \leftarrow e_2, s_1) \downarrow (\cdot, s_3\,[l \leftarrow v_2])}\ \mathit{assign}$$

$$\frac{}{(\lambda(x.e[x]), s) \downarrow (\lambda(x.e[x]), s)}\ \mathit{lambda}$$

$$\frac{(e_2, s_1) \downarrow (v_2, s_2) \quad (e_1, s_2) \downarrow (\lambda(x.e_3[x]), s_3) \quad (e_3[v_2], s_3) \downarrow (v_1, s_4)}{(e_1\ e_2, s_1) \downarrow (v_1, s_4)}\ \mathit{apply}$$

a function until the function is applied. Finally, the semantics is "eager:" the argument of an application is evaluated before the function is applied.

This semantics has a straightforward encoding in the meta-logic. The program primitives have corresponding term representations ($\mathtt{number}i$, $\mathtt{inl}(t)$, $\mathtt{inr}(t)$, $\mathtt{pair}(t_1; t_2)$, ...), and the value and evaluation judgments $\mathtt{value}(\mathtt{t})$ and $\mathtt{evalsto}(\mathtt{t}_1; \mathtt{s}_1; \mathtt{t}_2; \mathtt{s}_2)$. The translation of the rules for the value and evalsto judgments are also the normal translations (with second-order variables for the rule meta-variables, as in Chapter 2).

The interpretation of the state operations is more difficult, because the meta-logic does not provide a built-in construct for checking arbitrary inequalities. This affects two rules: the rule for reducing a dereference, and the rule for allocation. For example, we can encode one case ($l_1 = l_2$) of the dereference as the following reduction:

$$\mathtt{deref}(\mathtt{assign}(\mathtt{s}; \mathtt{l}; \mathtt{v}); \mathtt{l}) \longrightarrow \mathtt{v},$$

but the other case ($l_1 \neq l_2$) is inexpressible. Instead of coding the reduction rules directly, we will need to define the "side-conditions" for

Table 3.6: Encoding of state operations

deref(assign(s;l_2;v);l_1)	$\longrightarrow$	ifthenelse(eq(l_1;l_2);v;deref(s;l_1))
alloc(s;v)	$\longrightarrow$	pair(newlabel(s);newstate(s;v))
newlabel([])	$\longrightarrow$	first
newlabel(assign(s;l;v))	$\longrightarrow$	newlabel(s)
newlabel(newstate(s;v))	$\longrightarrow$	next(newlabel(s))
eq(first;first)	$\longrightarrow$	true
eq(first;next(l))	$\longrightarrow$	false
eq(next(l);first)	$\longrightarrow$	false
eq(next(l_1);next(l_2))	$\longrightarrow$	eq(l_1;l_2)
ifthenelse(true;t_1;t_2)	$\longrightarrow$	t_1
ifthenelse(false;t_1;t_2)	$\longrightarrow$	t_2

equality, inequality, and set operations directly in the meta-logic. We will take the following approach. First, the equality of labels will be expressed directly as a term:

$$[\![l_1 = l_2]\!] \longrightarrow \mathtt{eq}([\![\mathtt{l}_1]\!];[\![\mathtt{l}_2]\!]).$$

Finally, rather than encoding the judgment $\mathbf{dom}(s)$ directly, we introduce new terms newlabel(s) and newstate(s;) to represent the *results* an allocation operation. The *sequence* of labels returned by allocation will belong to the sequence first, next(first), next(next(first)), ... No interpretation is given to these locations beyond the equality judgment. Table 3.6 shows the interpretation of the state operations using these definitions.

This encoding deserves some discussion. The normal interpretation of the meta-logic is constructive: in general, the interpretation of an implication $A \longrightarrow B$ is that there is a computable function that will transform a proof of A into a proof of B. By defining equality on labels as a term, and defining the reductions, we essentially assert that equality of labels is computable. We could eliminate this assumption by not including the reductions for eq(l_1;l_2), but computability of pointer equality is a reasonable assumption.

3.2 Deriving a type system

At this point we have characterized the *evaluation* of simple imperative programs, and it would be reasonable to want to derive an automated evaluation procedure with the following specification: "any program evaluates to a value." Unfortunately, this is not *true*; what about the program that adds a number and a pair? In languages with run-time checking (like Lisp or Java), the program would generate a run-time error; in weakly-typed languages like C, the program would probably generate a compile-time warning, and if executed the result would be undefined; in strongly-typed languages like ML or Haskell, the program would be rejected at compile time.

We can adopt any of these approaches in our meta-logical interpretation: we can augment the programming language with *exceptions*, undefined values, or a type system. Since we are concerned most about specification, the choice naturally comes down to an issue of specification: can we determine which programs generate run-time errors, and which programs evaluate to a value? The reasoning techniques for this question are in normally in the realm of type systems, where well-typed programs are guaranteed not to generate run-time evaluation errors, and the natural course of action is to develop a type system for imperative programming in our framework.

Developing a type system can be a difficult task, especially in the presence of side-effects. To do it correctly, we need to build a model of program states and computation. We can develop it from first principles. Even better, we can build a *model* of imperative programs using the simple type theory we have already developed. If we can encode the programs in the type theory, we can "borrow" the types for interpreting imperative programs. This is an example of re-use; it is also an example of a relation between logics (where the imperative theory is *derived* from the type theory).

In MetaPRL, the derivation uses the module system notation of OCaml to state the derivation as a module functor. If `SimpleTypeTheory` is the module containing the definition of the type theory, and `SimpleImperativeLanguage` is the module containing the program syntax and operational semantics of the imperative language, the derivation is an instance of the following functor:

```
module SILofSTT (STT : SimpleTypeTheory) :
  SimpleImperativeLanguage.
```

The obligation of derivation is to give a justification for all of the primitive syntax and rules of the derived logic from the rules and definitions of the assumed logic. In this case, it means that we have to give

an encoding of imperative programs in terms of the functional syntax, and we have to derive all of the operational rules. In the MetaPRL system, this means that in a *proof* of SILofSIT, the semantic rules of SimpleImperativeLanguage are proof obligations, to be proved from the rules of SimpleTypeTheory.

We will perform this derivation in three parts:

1. derive an interpretation of Boolean values from STT (for use in the interpretation of locations),
2. develop an interpretation of locations and states,
3. prove the operational rules of SIL.

3.2.1 Developing the Boolean values

The definition of labels in the meta-logic (Table 3.6) uses a version of Boolean values `true` and `false`, and it will be convenient to derive them as a separate module (so they can be re-used later). We can interpret the Boolean values as a disjoint union *Unit* + *Unit* having two values, where we interpret `true` = *inl*(·) and `false` = *inr*(·). The definition of this module is shown in Table 3.7. The STT argument defines this module as an *extension* of the SimpleTypetheory module; the **define** statements define the syntax of Booleans in terms of the syntax in STT. The last part of the module contains *theorems*: the **rwthm** statements require proofs of reductions for the conditional, and the **thm** derive the inference rules for programs having type `bool`. The proofs (not listed) are a matter of unfolding the definitions and applying the rules of the type theory. For example, the derivation of `reduce_if_true` is a three-step process.

```
ifthenelse(true;t1;t2) ⟷ t1
BY rw (unfold_true then unfold_ifthenelse)
decide(inl(it);x.t1;y.t2) ⟷ t1
BY rw reduce_decide_inl
t1 ⟷ t1
BY axiom
```

Note that the module for Booleans is a *derived* module. With the exception of the new syntax definitions, the rules and rewrites of the Boolean module are proved from STT. This means that we can use the module in any context where the STT assumptions are valid, as we will see in the interpretation of locations and states.

Table 3.7: Defining a version of Boolean values

module BoolOfSTT (STT : SimpleTypeTheory) =
struct
  define unfold_bool : bool $\longleftrightarrow$ union(unit;unit)
  define unfold_true : true $\longleftrightarrow$ inl($\cdot$)
  define unfold_false : false $\longleftrightarrow$ inr($\cdot$)
  define unfold_ifthenelse :
    ifthenelse(t_1;t_2;t_3) $\longleftrightarrow$ decide(t_1;x.t_2;y.t_3)
  rwthm reduce_if_true : ifthenelse(true;t_1;t_2) $\longleftrightarrow$ t_1
  rwthm reduce_if_false : ifthenelse(false;t_1;t_2) $\longleftrightarrow$ t_2
  thm bool_type : $\Gamma \vdash$ type(bool)
  thm true_wf : $\Gamma \vdash$ true $\in$ bool
  thm false_wf : $\Gamma \vdash$ false $\in$ bool
  thm bool_elim :

$$\frac{\Gamma,\ \Delta[\mathsf{true}] \vdash T[\mathsf{true}] \quad \Gamma,\ \Delta[\mathsf{false}] \vdash T[\mathsf{false}]}{\Gamma,\ b{:}\mathsf{bool},\ \Delta[b] \vdash T[b]}$$

  thm bool_split_left :

$$\frac{\begin{array}{l}\Gamma,\ x{:}T_1[b],\ \Delta[x] \vdash b \in \mathsf{bool}\\ \Gamma,\ x{:}T_1[\mathsf{true}],\ \Delta[x] \vdash T_2[x]\\ \Gamma,\ x{:}T_1[\mathsf{false}],\ \Delta[x] \vdash T_2[x]\end{array}}{\Gamma,\ x{:}T_1[b],\ \Delta[x] \vdash T_2[x]}$$

  thm bool_split_right :

$$\frac{\Gamma \vdash b \in \mathsf{bool} \quad \Gamma \vdash T[\mathsf{true}] \quad \Gamma \vdash T[\mathsf{false}]}{\Gamma \vdash T[b]}$$

end

3.2.2 Interpreting locations and states

The interpretation of state is perhaps the most important choice in the derivation. The properties of states make it natural to interpret them as records: functions from labels to values. We'll need a little more: *allocation* requires the generation of a `new` label based on the labels already in the domain of the state. The locations themselves can be interpreted as the natural numbers, and we will choose a state interpretation where the state is a pair containing the "last" location allocated, and a record giving the state values. The definitions of the state operations are shown in Table 3.8.

`Bool, unfold_true, unfold_false, unfold_if:` The first step in the interpretation is to instantiate the `BoolOfSTT` module to be able to use the Boolean values in the interpretation of labels. This also brings us to the issue of naming: the `SIL` and `Bool` modules both have definitions for `true`, `false`, and the condition $\texttt{ifthenelse}(\texttt{t}_1;\texttt{t}_2;\texttt{t}_3)$. In cases where naming conflicts arise, names can be resolved in MetaPRL using the `ModuleName.TermName` syntax, where `ModuleName` is the name of the module in which the term was defined. In this case, the definition for `unfold_true` defines a term called `true` in the `SILofSTT` module, and it is defined directly as the Boolean term of the same name.

`unfold_first, unfold_next, unfold_eqlabel:` Since locations are numbers, the operations on locations correspond to operations on numbers, where the *first* location is 0, and successive locations are succeeding numbers.

`unfold_empty:` States are interpreted as *pairs*, where the first part is the *next* location to be allocated, and the second part is a *function* from labels to values. There are no assigned locations in the [] state, and any function is a valid implementation. This won't affect the interpetation because well-formed programs do not access locations that have not been allocated.

`unfold_deref:` The state value at location l is the *application* of the state function to argument l.

`unfold_assign:` The assignment operation $\texttt{assign}(\texttt{s};\texttt{l}_1;\texttt{v})$ returns a new state, with value v stored at location l_1. The function definition is a function that returns value v on label l_1, On labels l_2 that are not l_1 or 0, the original state value is returned.

Table 3.8: Interpretation of states and locations

```
module SILofSTT (STT : SimpleTypeTheory) =
struct
  module Bool = BoolOfSTT (STT)
  define unfold_true :  true ⟷ Bool.true
  define unfold_false :  false ⟷ Bool.false
  define unfold_if :  ifthenelse(t1;t2;t3) ⟷
    Bool.ifthenelse(t1;t2;t3)
  define unfold_first :  first ⟷ zero
  define unfold_next :  next(t) ⟷ succ(t)
  define unfold_eqlabel :  eq(l1;l2) ⟷ eq_int(l1;l2)
  define unfold_empty :  [] ⟷ pair(0;lambda(l.l))
  define unfold_deref :  deref(s;l) ⟷ apply(snd(s);l)
  define unfold_assign :  assign(s;l1;v) ⟷
    spread(s;l,f.pair(l;lambda(l2. ifthenelse(eq(l2;l1); )))
                                    v;
                                    deref(s;l2))
  define unfold_newlabel :  newlabel(s) ⟷ fst(s)
  define unfold_newstate :  newstate(s;v) ⟷
    spread(s;l,f.pair(succ(l); ifthenelse(eq(l2;l); ))
                                v;
                                deref(s;l2))
  define unfold_alloc :  alloc(s;v) ⟷
    pair(newlabel(s);newstate(s;v))
  rwthm reduce_thm1 :
    deref(assign(s;l2;v);l1) ⟶ ifthenelse(eq(l1;l2);
                                  v;
                                  deref(s;l1))
  ⋮
```

`unfold_newlabel, unfold_newstate, unfold_alloc:` In any state, the next location is given as the first part of the state-pair. The `newstate(s;v)` updates the next-location value, and it updates the state-function to return v on the newly-allocated location.

`reduce_thm1, ..` The state operations (Table 3.6) can now be derived as *theorems* in this interpretation. In all cases, the proof is simply a matter of unfolding the definitions.

3.2.3 Interpreting the programs

To interpret the programs, we will need to "thread" the state explicitly. The translation is fairly mechanical: each expression becomes a state transformer, and values are computed by *applying* an expression to a state argument. The value computed by a program is a value/state pair. For example, the interpretation of the numbers is a function that returns a number without modifying the state:

$$\begin{array}{rcl} 0 & \longrightarrow & \texttt{lambda}(s.\texttt{pair}(\texttt{zero};s)) \\ 1 & \longrightarrow & \texttt{lambda}(s.\texttt{pair}(\texttt{succ}(\texttt{zero});s)) \\ & \vdots & \end{array}$$

In this model the *addition* function computes the numbers of its two subexpressions:

$$\begin{array}{rcll} \texttt{add}(e_1;e_2) & \longrightarrow & \lambda(s_1. & \textbf{match}\, e_1\ s_1\ \textbf{with}\ (v_1,s_2) \rightarrow \\ & & & \quad \textbf{match}\, e_2\ s_2\ \textbf{with}\ (v_2,s_3) \rightarrow \\ & & & \quad\quad ((v_1+v_2),s_3)). \end{array}$$

The complete list of these translations in given in Table 3.9. For clarity, the terms are displayed using normal mathematical notation, but it should be understood that the terms are being defined in the `SILofSTT` module, and the definitions are defined from terms in the `STT` module. The first definition, for `expr(v)`, represents a conversion from a value in `STT` to a value in `SIL`. It is used in the `decide`, `spread`, `ref`, and `apply` definitions to produce expressions that can be used as arguments to substitution. The `ref`, `deref`, and `assign` are the only operations that access the state.

3.2.4 Deriving the natural semantics

So far, we have just given *definitions* of the `SIL` programming language in term of the `STT` type theory. How do we know we have a faithful

Table 3.9: Program interpretations

define unfold_expr : expr(v) $\longleftrightarrow$ λ(s.(v, s))
define unfold_number : $i \longleftrightarrow \lambda(s.(i, s))$
define unfold_add : $(e_1 + e_2) \longrightarrow$
  $\lambda(s_1.$ **match**$(e_1\ s_1)$ **with** $(v_1, s_2) \rightarrow$
    match$(e_2\ s_2)$ **with** $(v_2, s_3) \rightarrow$
      $((v_1 + v_2), s_3))$
define unfold_if : **if** e_1 **then** e_2 **else** $e_3 e_4 \longleftrightarrow$
  $\lambda(s_1.$ **match**$(e_1\ s_1)$ **with** $(v_1, s_2) \rightarrow$
    match$(e_2\ s_2)$ **with** $(v_2, s_3) \rightarrow$
    if $v_1 = v_2$ **then** $(e_3\ s_3)$ **else** $(e_4\ s_4))$
define unfold_inl : $inl(e) \longleftrightarrow$
  $\lambda(s_1.$ **match**$(e_1\ s_1)$ **with** $(v_1, s_2) \rightarrow$
    $(inl(v_1), s_2))$
define unfold_decide :
  (**match** e_1 **with** $inl(x) \rightarrow e_2[x] \mid inr(y) \rightarrow e_3[y]) \longleftrightarrow$
  $\lambda(s.$ **match**$(e_1\ s_1)$ **with** $(v_1, s_2) \rightarrow$
    match v_1 **with**
     $inl(x) \rightarrow (e_2$[expr(x)] $s_2)$)
    | $inr(y) \rightarrow (e_3$[expr(y)] $s_2)$
define unfold_pair : $(e_1, e_2) \longleftrightarrow$
  $\lambda(s_1.$ **match**$(e_1\ s_1)$ **with** $(v_1, s_2) \rightarrow$
    match$(e_2\ s_2)$ **with** $(v_2, s_3) \rightarrow$
    $(\mathtt{pair}(v_1; v_2), s_3)$
define unfold_spread : (**match** e_1 **with** $(x, y) \rightarrow e_2[x; y]) \longleftrightarrow$
  $\lambda(s_1.$ **match**$(e_1\ s_1)$ **with** $(v_1, s_2) \rightarrow$
    match v_1 **with** $(x, y) \rightarrow (e_2$[expr(x);expr(y)] $s_2)$
define unfold_ref : **ref** $e \longleftrightarrow$
  $\lambda(s_1.$ **match**$(e\ s_1)$ **with** $(v_1, s_2) \rightarrow$
    $(\mathtt{expr}(\mathtt{newlabel}(s_2)), \mathtt{newstate}(s_2; v_1))$
define unfold_deref : $!e \longleftrightarrow$
  $\lambda(s_1.$ **match**$(e\ s_1)$ **with** $(v_1, s_2) \rightarrow$
    $(\mathtt{expr}(\mathtt{deref}(s_2; v_1)), s_2)$
define unfold_assign : $e_1 \leftarrow e_2 \longleftrightarrow$
  $\lambda(s_1.$ **match**$(e_1\ s_1)$ **with** $(v_1, s_2) \rightarrow$
    match$(e_2\ s_2)$ **with** $(v_2, s_3) \rightarrow$
    $(\cdot, \mathtt{assign}(s_2; v_1; v_2))$
define unfold_fun : $\lambda(v.e[v]) \longleftrightarrow$ expr(λ(v.e[v]))
define unfold_apply : $e_1\ e_2 \longleftrightarrow$
  $\lambda(s_1.$ **match**$(e_2\ s_1)$ **with** $(v_2, s_2) \rightarrow$
    match$(e_1\ s_2)$ **with** $(v_1, s_3) \rightarrow$
    $(v_1\ \mathtt{expr}(v_2))\ s_3$

model? The accuracy of our model comes down to a derivation of the semantics of the imperative programs: can we intepret the rules for reduction and show that they are valid in the type theory? To begin this process, we first have to interpret the `value(e)` and $(e, s_1) \downarrow (v, s_2)$ judgments in the type theory, and to do that we have to understand what the judgments *mean*.

Intuitively, we think of a *value* as a program that does not access the state of a program. A value should be a program that evaluates to the same result in *any* state. For the $(e, s_1) \downarrow (v, s_2)$ judgment, we mean that there is some sequence of reductions that will reduce an program e in state s_1 to an equivalent program v, producing a new state s_2 and v *is a value*. If our intution is right, we should be able to derive the natural semantics with the following definitions of the judgments:

$$\begin{array}{rcl} \mathit{value}(v) & \longrightarrow & (v \longleftrightarrow \lambda(s.(\mathbf{fst}(v\ []), s))) \\ (e, s_1) \downarrow (v, s_2) & \longrightarrow & ((e\ s_1 \longleftrightarrow v\ s_2) \wedge \mathit{value}(v)). \end{array}$$

For $\mathit{value}(v)$, we mean that the value is equivalent to the program that produces the same result in the *empty* state. The $(e, s_1) \downarrow (v, s_2)$ judgment means that programs e and v produce the same results in their respective states, and v is a value.

These judgments are not currently expressible in the type theory, because there is no equality judgment. To reason about these judgments, we will need to add an equivalence judgment to STT as follows:

$$\frac{}{\Gamma \vdash t \longleftrightarrow t}\ \mathit{equiv}$$

$$\frac{\Gamma,\ x : T_1[a],\ \Delta[x] \vdash a \longleftrightarrow b \quad \Gamma,\ x : T_1[b],\ \Delta[x] \vdash T_2[x]}{\Gamma,\ x : T_1[a],\ \Delta[x] \vdash T_2[x]}\ \mathit{hyp\text{-}subst}$$

$$\frac{\Gamma \vdash a \longleftrightarrow b \quad \Gamma \vdash T[b]}{\Gamma \vdash T[a]}\ \mathit{goal\text{-}subst.}$$

As expected, with these definitions we can derive all of the natural semantics; and once again, the proofs are mainly a matter of unfolding definitions. To illustrate the process, let's derive the proof of evaluation for function application. The proof itself is listed in Table 3.10. The first step is to unfold the definitions of `evalsto` and `apply`, and then apply the standard set of STT reductions. In the second step, we apply the rewrite $e_2\ s_1 \longleftrightarrow v_2\ s_2$ (the evaluation part of the `evalsto` judgment), and in the third step we apply the rewrite rewrite $v_2 \longleftrightarrow \lambda(s.(\mathbf{fst}(v\ []), s))$ (from the `value` part of the `evalsto` judgment), and then reduce. The fourth step is similar, which reduces the goal to an assumption.

Table 3.10: Evaluation of application

$$\frac{\Gamma \vdash (e_2, s_1) \downarrow (v_2, s_2) \quad \Gamma \vdash (e_1, s_2) \downarrow (\lambda(v.e_3[v]), s_3) \quad \Gamma \vdash (e_3[v_2], s_3) \downarrow (v_3, s_4)}{\Gamma \vdash (e_1\ \ e_2, s_1) \downarrow (v_3, s_4)}$$

```
BY unfold_evalsto then unfold_apply then reduce (1)
```
$(\mathbf{match}(e_2\ \ s_1)\ \mathbf{with}\ (v_2, s_2) \rightarrow$
$\mathbf{match}(e_1\ \ s_2)\ \mathbf{with}\ (v_1, s_3) \rightarrow$
$(v_1\ \ v_2)\ \ s_3) \downarrow v_3\ \ s_4$
```
BY rewrite with (e2,s1) ↓ (v2,s2) and reduce (2)
```
$(\mathbf{match}(e_1\ \ s_2)\ \mathbf{with}\ (v_1, s_3) \rightarrow$
$(v_1$ `expr(`$\mathbf{fst}$`(v2 []))) s3 ↓ v3 s4`
```
BY fold expr v2 (3)
```
$(\mathbf{match}(e_1\ \ s_2)\ \mathbf{with}\ (v_1, s_3) \rightarrow$
$(v_1\ \ v_2)\ \ s_3 \downarrow v_3\ \ s_4$
```
BY rewrite with (e1,s1) ↓ (λ(v.e3[v]),s3) and reduce (4)
```
$e_3[v_2]\ \ s_3 \downarrow v_3\ \ s_4$
```
BY axiom (5)
```

3.2.5 Enhancing the type theory

Now that we have produced and derived this model, what does it buy us? The answer is this: the type theory provides typing judgments and program equivalence reasoning that applies to all of the imperative programs. Remember the dependent typing of the remainder? That theorem can be re-used in a verification proof of a corresponding imperative remainder. In other words, by deriving a logic, we can re-use the theorems of the assumed logic.

That doesn't mean that the derivation is non-trivial. In this case, our primary difficulty is in the typing of program state—a problem that lies at the heart of the issues with side-effects.

The operational description of state presents it as a function from labels to values. Another way to think of it is that the function represents a *record*, a data structure common to many programming languages. Records have an operational description just like the state.

For a record $r = \left\{ l_i = v_i^{\,i \in \{1,\ldots,n\}} \right\}$:

$$r.l_j \ \longrightarrow\ v_j \qquad \text{(projection)}$$

$$r\left[l_j \leftarrow v\right] \ \longrightarrow\ \left\{ l_i = v_i^{\,i \in \{1,\ldots,n\}-\{j\}}, l_j = v \right\} \qquad \text{(update)}$$

Table 3.11: Interpretation of records in STT

$$\begin{array}{rcl}
\left\{l_i = v_i^{i\in\{1,\ldots,n\}}\right\} & = & \lambda l.\,\textbf{if}\ l = l_1\ \textbf{then}\ v_1 \\
& & \textbf{else if}\ l = l_2\ \textbf{then}\ v_2 \\
& & \vdots \\
& & \textbf{else if}\ l = l_{n-1}\ \textbf{then}\ v_{n-1} \\
& & \textbf{else}\ v_n \\
\\
\left\{l_i\colon T_i^{i\in\{1,\ldots,n\}}\right\} & = & l\colon \mathit{Label} \rightarrow \textbf{if}\ l = l_1\ \textbf{then}\ T_1 \\
& & \qquad\qquad \textbf{else if}\ l = l_2\ \textbf{then}\ T_2 \\
& & \qquad\qquad \vdots \\
& & \qquad\qquad \textbf{else if}\ l = l_{n-1}\ \textbf{then}\ T_{n-1} \\
& & \qquad\qquad \textbf{else}\ T_n
\end{array}$$

The typing of records has the form $R = \left\{l_i\colon T_i^{i\in\{1,\ldots,n\}}\right\}$, meaning that any record r with type R has the field types $r.l_j \in T_j$. Records can be interpreted directly in the type theory as dependent function types, as shown in Table 3.11.

Records have a natural subtyping property: a record with more fields implements a record with fewer. To exploit this property, we will need to add two judgments, and a new type to the type theory, for which the rules are shown in Table 3.12. The first is just a subtype judgment $T_1 \le T_2$; if the judgment is true, then all programs with type T_1 also have type T_2. It will also be useful to add an equality judgment. This judgment can be defined in terms of membership, and a substitution principle. In addition, two functions should be equal when they produce equal values on equal arguments. The last type we will need the the intersection $\bigcap x\colon A.B[x]$, which contains those programs that have type $B[v]$ for *any* $v \in A$.

These types provide the basis for deriving a set of rules for reasoning about records. The subtyping property of records can be derived from an argument by function extensionality. The other property of interest is the *intersection* of records types, which corresponds to *concatenation* of the types (where $T_1 \equiv T_2$ is defined as $T_1 \le T_2 \wedge T_2 \le T_1$):

$$\Gamma \vdash \bigcap i.\{1,\ldots,n\{l_i\colon T_i\} \equiv \left\{l_i\colon T_i^{i\in\{1,\ldots,n\}}\right\}.$$

The proofs of both of these properties follows from function extensionality.

Table 3.12: Additional judgments in the type theory

$$t \in T \longleftrightarrow (t = t \in T) \quad \textit{(unfold_member)}$$

$$\frac{\Gamma \vdash t_1 = t_2 \in T \quad \Gamma,\ v{:}T \vdash C[v] \in \textit{Type} \quad \Gamma \vdash T[t_2]}{\Gamma \vdash C[t_1]}\ \textit{goal_subst}$$

$$\frac{\begin{array}{l}\Gamma,\ x{:}T_1[t_1],\ \Delta[x] \vdash t_1 = t_2 \in T_2 \\ \Gamma,\ x{:}T_1[t_1],\ \Delta[x],\ v{:}T_2 \vdash T_2[v] \in \textit{Type} \\ \Gamma,\ x{:}T_1[t_2],\ \Delta[x] \vdash C[x]\end{array}}{\Gamma,\ x{:}T[t_1],\ \Delta[x] \vdash C[x]}\ \textit{hyp_subst}$$

$$\frac{\begin{array}{l}\Gamma \vdash x{:}C \to D[x] \in f_1 \\ \Gamma \vdash x{:}E \to F[x] \in f_2 \\ \Gamma \vdash A \in \textit{Type} \\ \Gamma,\ v{:}A \vdash f_1\ v = f_2\ v \in B[v]\end{array}}{\Gamma \vdash f_1 = f_2 \in x{:}A \to B[x]}\ \textit{extensionality}$$

$$\frac{\Gamma \vdash T_1 \in \textit{Type} \quad \Gamma,\ v{:}T_1 \vdash v \in T_2}{\Gamma \vdash T_1 \le T_2}\ \textit{subtype}$$

$$\frac{\Gamma \vdash A \in \textit{Type} \quad \Gamma,\ x{:}A \vdash t \in B[x]}{\Gamma \vdash t \in \bigcap x{:}A.B[x]}\ \textit{isect_member}$$

$$\frac{\begin{array}{l}\Gamma,\ v{:}\bigcap x{:}A.B[x],\ \Delta[v] \vdash a \in A \\ \Gamma,\ v{:}\bigcap x{:}A.B[x],\ \Delta[v],\ y{:}x \in B[a] \vdash C[v]\end{array}}{\Gamma,\ v{:}\bigcap x{:}A.B[x],\ \Delta[v] \vdash C[v]}\ \textit{isect_elim}$$

Chapter 4

Modular composition of reactive systems

For the final part of this introduction, we show give a more extensive example showing how to use the type theory of the last chapter to implement a formal framework for specifying and verifying modular systems. The focus of this chapter is on a method for scaling the methodology to large numbers of modules and protocols. We present formal operators, for composing specifications and re-using verification proofs.

The specific example we use is the Ensemble group communication system [Hay97, Hay98]. Ensemble is a highly modular group communication system that eases the task of distributed programming by providing configurable implementations of critical distributed algorithms like failure detection and recovery, while giving precise guarantees for message ordering and delivery. This is ideal from the perspective of reliability: the critical distributed algorithms are cleanly isolated and identified. Formal verification of these protocols applies to all applications that use Ensemble.

The Ensemble architecture is designed around layered protocol stacks, composed from a collection of protocol layers that provide a roughly orthogonal set of primitives. Each layer is about 300 lines of code in Objective Caml [RV97]. Currently there are about 50 different layers that can be composed to implement thousands of useful protocols. Clearly, any feasible formal framework must capitalize on modularity and re-use.

We present a formal framework for specifying and verifying modular systems like Ensemble based on the I/O automata of Lynch and Tuttle [Lyn96, LT89]. I/O automata provide an ideal specification language in some respects. They can specify modular components that range from the concrete OCaml layer code to abstract distributed services. However, the automata defined by Lynch and Tuttle do not, in

general, allow formal theorems to be re-used as protocols are modified. For example, a FIFO protocol may have the informal specification, "all messages are delivered in the order in which they were sent." When a total ordering layer is added to the protocol, it adds the additional property, "all processes receive messages in the same order." In the standard formalization of I/O automata, the TOTAL protocol is different from the FIFO protocol, and the FIFO property must be re-proved after the total-ordering property is added.

We introduce a type-theoretic foundation for I/O automata using the formal system described in the last chapter for reasoning about imperative programs. Our formalization preserves safety properties as automata are composed. The definition of composition is based on the *intersection* of automata: if A and B are automata, and A has safety property P (written $A \vDash \mathit{always}\, P$), then $A \cap B \vDash \mathit{always}\, P$. In fact, the sentence $(A \vDash \mathit{always}\, P) \Rightarrow (A \cap B \vDash \mathit{always}\, P)$ is a formal *theorem* in our framework. The intersection of automata corresponds to inheritance in object-oriented languages. Our paradigm advocates the construction of specifications from a library of small, easily-understood components, using compositional intersection to construct service specifications.

In the remainder of the chapter, we cover the formal framework in more detail. In Section 4.1 we develop the framework by adding intersections to "standard" I/O automata and giving the type theoretic interpretation. In Section 4.6 we present the specification of virtual synchrony, a major service provided by group communication systems. Our work shares many of the features of conjoined TLA specifications described by Abadi and Lamport [AL95]; we discuss related work in Section 4.7.

4.1 I/O automata

General I/O automata are instances of *labeled transition systems*. An automaton consists of a set of states Q; a set of labels A, called *actions*, that label each of the transitions of the automaton; a set of *initial* states i; and a set of allowable transitions t. In type theory, an automaton is described by the following type:[1]

[1] We use the notation $\left|\begin{array}{l} P \\ \text{METAPRL} \end{array}\right.$ to denote formulas P expressed in MetaPRL.

$$
\begin{array}{|lll}
\mathit{Auto} & \equiv & \quad Q\colon \mathit{Type} \\
 & & \times\ A\colon \mathit{Type} \\
 & & \times\ i\colon Q \to \mathit{Prop} \\
 & & \times\ t\colon (Q \times A \times Q) \to \mathit{Prop} \\
\textsc{MetaPRL} & &
\end{array}
$$

The initial states i are a subset of the possible states Q, defined using a predicate $Q \to \mathit{Prop}$, where *Prop* is the set of propositions (for simplicity, this can be thought of as a Boolean value). The transition definition is similar. Note that the states and actions may be types with an infinite number of elements.

I/O automata are often defined using stylized pseudo-code. The *actions* correspond to "significant" external events of the system being specified. For example, a simple specification of a FIFO protocol is shown below. The FIFO actions are SEND(m) and RECV(m), where m ranges over some message type $\mathcal{M}$. The automaton contains two queues that save the messages *sent* and *received*. When a message is sent, with the transition labeled SEND(m), the message is added to the *sent* list of messages. Messages are received with the transition labeled RECV(m). A message m can be received when there are more messages in the *sent* queue than the *received* queue and the next message in *sent* is m; the effect is to add the message to the *received* queue.

FIFO

Actions: SEND(m), RECV(m), for $m \in \mathcal{M}$
State: $\mathit{sent} \in \mathcal{M}$ *List*, initially empty,
$\mathit{received} \in \mathcal{M}$ *List*, initially empty

SEND(m)
Eff: append m to *sent*

RECV(m)
Pre: $|\mathit{received}| < |\mathit{sent}|$
$\mathit{sent}[|\mathit{received}| + 1] = m$
Eff: append m to *received*

In general, an automaton defines a set of *executions* $q_0, a_0, q_1, a_1, \ldots$, where each triple q_i, a_i, q_{i+1} is a valid transition of the automaton. It also defines a set of *traces*, which are the actions of the valid executions. If A and B are automata, and the traces of A are a subset of the traces of B, we say that A *implements* B. An *invariant* is a predicate that is true of all states in all executions of an automaton. The FIFO automaton defines that invariant that the *received* queue is always a prefix of the *sent* queue: FIFO $\vDash$ *always received* $\leq$ *sent*.

So far, this presentation of I/O automata adheres to the original form[2] of Lynch and Tuttle. Now suppose we wish to define a form of composition that preserves safety properties, which include state invariants and properties of individual traces of the automaton. The obvious choice for composition is the *intersection* of their traces. If M and M' are automata, their intersection has states that are states both of M and M', and traces that belong both to M and M'. If $M = (Q, A, i, t)$ and $M' = (Q', A', i', t')$, we define their intersection as follows:

$$
\begin{array}{|ll}
M \cap M' \equiv & (\ Q \cap Q', \\
 & \quad A \cap A', \\
 & \quad \lambda s.(i(s) \wedge i'(s)), \\
 & \quad \lambda q_1, a, q_2.t(q_1, a, q_2) \wedge t'(q_1, a, q_2)). \\
\text{METAPRL} &
\end{array}
$$

With this definition, we have the following theorem.

Theorem 1 *For any automata M and M, and state predicate P, if $M \vDash$ always P, then $M \cap M' \vDash$ always P.*

The proof is straightforward, since any execution of $M \cap M'$ is also an execution of M. In the degenerate case, if the state spaces Q and Q' do not have any elements in common, the intersection $M \cap M'$ is the empty automaton; it has no executions or traces. In our framework, we address this problem by including "all" state variables and actions in every automata, but leaving almost all of them unspecified, as shown in the following section.

4.2 Stylized automata

The pseudo-code suggests a *style* of automata definition that can be used to guide their construction. In the pseudo-code, the actions have a *label* (like SEND or RECV), and a *value* (like $m \in \mathcal{M}$) that is associated with the label. Given a type of labels *Label*, the set of stylized actions can be defined as the dependent sum $Action \equiv \Sigma l{:}Label.F_A(l)$ where $F_A \in Label \rightarrow Type$ is a type-function that defines the type of the value associated with each label. For example, the actions of the FIFO automaton are defined by the following Σ-type:

[2]Lynch and Tuttle also labels actions as being *input*, *output*, or *internal* to give a little more semantic content, but we find this labeling unnecessary.

$\Sigma l\colon \mathit{Label}.(\textbf{if } l = \text{“SEND”} \vee l = \text{“RECV”} \textbf{ then } \mathcal{M} \textbf{ else } \mathit{Top})$.
MetaPRL

If the action label is SEND or RECV, the action *value* belongs to the type of messages $\mathcal{M}$. Otherwise, the data part is totally unspecified (the type *Top* is the type containing all terms, defined as the degenerate intersection $\bigcap_{x \in \mathit{Void}} .x$).

The stylized states are defined similarly. A state has a set of named variables (we can use the *Label* type), and each variable has a value. The most natural type is the record type. For example, the FIFO state can be represented by the record $\{\mathit{sent}\colon \mathcal{M}\ \mathit{List}; \mathit{received}\colon \mathcal{M}\ \mathit{List}\}$. Records are dependent-function spaces that map variable names to values, and the state type is defined as $\mathit{State} = \Pi l\colon \mathit{Label}.F_S(l)$ for some type-function $F_S \in \mathit{Label} \to \mathit{Type}$. The FIFO state has the following definition:

$\Pi l\colon \mathit{Label}.(\textbf{if } l = \text{“}\mathit{sent}\text{”} \vee l = \text{“}\mathit{received}\text{”} \textbf{ then } \mathcal{M}\ \mathit{List} \textbf{ else } \mathit{Top})$.
MetaPRL

Again, the state is *unspecified* on labels that are not in {“*sent*”, “*received*”}. This results in the main type-theoretic principle we use to avoid degeneracy: record *intersection* corresponds to record *concatenation*.

$\{l_1\colon T_1; l_2\colon T_{2a}\} \cap \{l_2\colon T_{2b}; l_3\colon T_3\} \equiv \{l_1\colon T_1; l_2\colon T_{2a} \cap T_{2b}; l_3\colon T_3\}$.
MetaPRL

The intersection property for the Σ type also has the concatenation property. When two automata are intersected, their state variables and action signatures are concatenated. This simple feature captures the object-oriented nature of our framework: the intersection of automata corresponds to *inheritance* in object-oriented languages.

4.3 Type-theoretic interpretation

The pseudo-code can be represented in the type-theory by separating the pseudo-code into the four parts of an automaton. The actions define a Σ type, the state variables define a record (a Π type), and the initial values of the state variables specify the initial conditions. The transition definitions define a transition relation: for an automata with state type Q, the transition definition $L(x)$ `Pre` $P(q,x)$ `Eff:` $\mathit{expr}(q,x)$ for $x \in T$ defines a conjunctive clause in the transition definition as follows:

$\forall x \in T.(a = L(x) \Rightarrow P(q_1,x) \wedge (q_2 = \mathit{expr}(q_1,x) \in Q))$.
MetaPRL

That is, the transition (q_1, a, q_2) is in the transition relation only if, whenever the action a is $L(x)$ for some $x \in T$, the precondition P holds on the initial state q_1, and the result of the effect is equal to the final state q_2 *on the variables specified in the state* Q. Other variables are unconstrained. The `Eff:` clause of a transition definition is optional; if it is omitted, the equality condition on q_2 is omitted in the type-theoretic interpretation, allowing the state to change arbitrarily. The formal definition of the FIFO automaton is the following program:

$$
\begin{array}{l}
Q = \Pi l \in \mathit{Label}.(\textbf{if } l = \text{“}\mathit{sent}\text{”} \vee l = \text{“}\mathit{received}\text{”} \textbf{ then } \mathcal{M}\ \mathit{List} \textbf{ else } \mathit{Top}),\\
A = \Sigma l \in \mathit{Label}.(\textbf{if } l = \text{“SEND”} \vee l = \text{“RECV”} \textbf{ then } \mathcal{M} \textbf{ else } \mathit{Top}),\\
i = \lambda q.(q.\mathit{sent} = [\,] \wedge q.\mathit{received} = [\,]),\\
t = \lambda q_1, a, q_2.\quad \forall m \in \mathcal{M}.(a = \text{SEND}(m)\\
\qquad\qquad \Rightarrow\ (q_2.\mathit{sent} = m \text{ appended to } q_1.\mathit{sent}\\
\qquad\qquad \wedge\ q_2.\mathit{received} = q_1.\mathit{received}))\\
\qquad \wedge\ \forall m \in \mathcal{M}.(a = \text{RECV}(m)\\
\qquad\qquad \Rightarrow\ (|q_1.\mathit{received}| < |q_1.\mathit{sent}|\\
\qquad\qquad \wedge\ q_1.\mathit{sent}[|q_1.\mathit{received}| - 1] = m\\
\qquad\qquad \wedge\ q_2.\mathit{received} = m \text{ appended to } q_1.\mathit{received}\\
\qquad\qquad \wedge\ q_2.\mathit{sent} = q_1.\mathit{sent}))
\end{array}
$$

METAPRL

4.4 Using the automata

Now, how do we use these automata? To give an example, let's modify the FIFO automaton so that no more than 100 messages are ever delivered. We construct a new automaton with the same state variable *received* used in the FIFO automaton, and we include a precondition on the RECV(m) action.

FINITE_FIFO	
`Actions:`	RECV(m) `for` $m \in \mathcal{M}$
`States:`	$\mathit{received} \in \mathcal{M}\ \mathit{List}$, `initially` []
RECV(m)	
`Pre:`	$\lvert\mathit{received}\rvert < 100$

The new automaton is not too useful by itself: it allows arbitrary state changes in the effect for RECV(m). When combined with the FIFO automaton, however, the new machine FIFO $\cap$ FINITE_FIFO has the *same* transitions and states of the FIFO automaton, except for the transition RECV(m), which has the following definition:

RECV(m)
Pre: $|received| < |sent|$
$sent[|received| - 1] = m$
$|received| < 100$
Eff: append m to $received$

The FIFO invariant $received \leq sent$ remains true in the new automaton, but now we can also prove that the received queue never contains more than 100 messages.

4.5 Operations on automata

There are some drawbacks to the automata we have just presented. First, every state machine shares the same state space—in essence, all state variables are global. We use this feature during composition (like the FINITE_FIFO example), but there is no notion of "public" or "private" variables, where private variables do not become shared during composition. Second, the action space is global as well, so it is not possible to re-use an automaton in multiple specifications with different action namings.

Both of these problems can be addressed by introducing *renaming* operations. We use the following informal notation, given an arbitrary renaming function $R \in Label \to Label$ on *labels*, and an automaton M:

M, renaming actions $l(x)$ to $R(l)(x)$
M, renaming variables v to $R(v)$.

When actions are renamed, the action type and the transition definitions must be modified. Consider the following stylized machine M:

$M = (Q = \Pi l \in Label.F_S(l), A = \Sigma l \in Label.F_A(l), i, t)$.
METAPRL

The action renaming "*M, renaming actions* $L(x)$ *to* $R(L)(x)$" defines the renamed machine M' shown below.

$$
\begin{array}{lllll}
M' = (& Q' = & \Pi l \in Label.F_S(l), & & \\
& A' = & \Sigma l \in Label.\left(\bigcap_{\{l' \in Label \mid R(l') = l\}} .F_A(l')\right) & & \\
& i' = & i, & & \\
& t' = & \lambda q_1, l(x), q_2. \textbf{ if } & \exists l' \in Label.(R(l') = l) \textbf{ then} & \\
& & & t(q_1, l'(x), q_2) & \\
& & \textbf{else} & & \\
& & & q_1 = q_2 \in Q') &
\end{array}
$$

METAPRL

Since R is an arbitrary renaming function, it may map several labels to a single value. The action type associated with label l in A' is the *intersection* of all the action types $F_A(l')$ for any label l' in the inverse image $R^{-1}(l)$. If $R^{-1}(l)$ is nonempty, the transition definition allows *any* of the transitions $l'(x)$ for any $l' \in R^{-1}(l)$. Otherwise, the value type on label l in A' degenerates to *Top*, and a transition is allowed only if it leaves the state unchanged. From this definition, we get the following formal meta-theorem for safety properties P:

Theorem 2 *For any automaton M, renaming function R, and state predicate P, if $M \vDash$ always P then $(M,$ renaming actions $l(x)$ to $R(l)(x)) \vDash$ always P.*

The state renaming operation is similar. The states, initial state predicate, and transition relation must be modified. The following machine defines the renamed machine $M'' =$ *m, renaming variables v to $R(v)$*:

$$
\begin{array}{|lcll}
M'' & = & (\ Q'' & = & \Pi l \in \mathit{Label}.\left(\bigcap_{\{l' \in \mathit{Label} \mid R(l')=l\}}.F_S(l')\right), \\
 & & A'' & = & \Sigma l \in \mathit{Label}.F_A(l), \\
 & & i'' & = & (i \circ R) \\
 & & t'' & = & \lambda q_1, a, q_2.t(q_1 \circ R, a, q_2 \circ R)) \\
\multicolumn{4}{l}{\textsc{MetaPRL}}
\end{array}
$$

This definition induces a renaming operation on state predicates P, providing the following formal meta-theorem:

Theorem 3 *For any automaton M, renaming function R, and state predicate P, if $M \models always\,P$, then $(M, renaming\ variables\ v\ to\ R(v)) \models always(\lambda q.P(q \circ R))$.*

4.6 Specifying Ensemble Virtual Synchrony

To demonstrate the use of our new version of I/O automata, we present the specification of Ensemble Virtual Synchrony (EVS). In the EVS model, processes join together to form *views*, which are sets of processes that vary over time as processes join and leave. Views are thought of informally as active multicast domains; when a process fails, or when the network becomes partitioned, a view may be split into several smaller views. When processes are created, or when the network heals, multiple views may merge into one. Each process has its own version of the view it is in. When a process replaces its view $view_1$ with a new view $view_2$, we say the process *installs* view $view_2$. Virtual synchrony provides the following informal properties on *views* and messages:

EVSsingle At any time a process belongs to exactly one view.

EVSself If a process installs a view, it belongs to the view.

EVSvieworder At any process, views are installed in increasing order of view identifier.

EVSnonoverlap If two processes install the same view, their previous views are either the same or they are disjoint.

EVSviewmessage All delivered messages are delivered in the view in which they were sent.

EVSfifo Messages between any two processes in a view are delivered in FIFO order.

EVSsync Any two process that install a view v_2, both with preceding view v_1, deliver the same messages in view v_1.

We can categorize these properties in three parts: message delivery (EVSfifo, EVSviewmessage), view properties (EVSsingle, EVSself, EVSvieworder, EVSnonoverlap), and message/view properties (EVSsync).

We can specify the message delivery properties using the FIFO automaton. Let $VID \subseteq Label$ be the type of views, and let $PID \subseteq Label$ be the type of processes. We implement the *Label* type as a recursive type including names and closed under pairing (which we used for subscripting by process and view), to get the following automaton:

EVS_FIFO = $\bigcap_{v \in VID} \bigcap_{p,q \in PID}$ ·
(FIFO renaming
action SEND(m) to EVS-SEND$_{p,v}(m)$
action RECV(m) to EVS-RECV$_{q,p,v}(m)$
variable *sent* to $sent_{p,v}$
variable *received* to $received_{q,p,v}$)

METAPRL

The EVSfifo property follows trivially from the FIFO automaton. The EVSviewmessage property follows because FIFO channels are only established within views (they do not cross views).

For the *view* properties, we introduce a new action EVS-NEWVIEW$_p(v)$, which delivers the view $v \in VID \times PID$ Set to process $p \in PID$. We refer to the view identifier as $v.id$ and the set of processes that belong to the view as $v.set$. We also need to introduce a history variable $all\text{-}viewids_p$, for each process $p \in PID$, containing the set of views ever delivered to process p. Several derived variables are defined in terms of $all\text{-}views_p$:

- $current\text{-}view_p$ is the view in $all\text{-}views_p$ with the largest identifier.
- $pred\text{-}view_{p,v}$ is the view in $all\text{-}views_p$ with identifier strictly smaller than $v.id$ if such a view exists, otherwise $\perp$, where $\perp \notin VID$ is a constant.

Given these definitions the view part of EVS is shown below. We define a new term mechanism to provide the high-level syntax used in this description.

EVS_VIEW

State: for each $p \in PID$:
$all\text{-}viewids_p \in View$ Set, initially $\{v_p\}$

EVS-NEWVIEW$_p(v)$
Pre: let $v' = current\text{-}view_p$ in
$v.id > v'.id$ ①
$p \in v.set$ ②
$\forall q \in v.set.$ ③
if $pred\text{-}view_{q,v} \neq \perp$ then
$pred\text{-}view_{q,v} = v' \vee pred\text{-}view_{q,v}.set \cap v'.set = \{\}$
Eff: $all\text{-}views_p = all\text{-}views_p \cup \{v\}$

METAPRL

The precondition for EVS-NEWVIEW$(v)_p$ has several parts, one for each of the view properties. Part ① guarantees that views are delivered in ascending order (EVSvieworder), and the definition of $current\text{-}view_p$

defines exactly one current view per process (EVSsingle). Part ② ensures that a process belongs to all its view (EVSself). Part ③ sets up the condition on previous views: if another process q has installed view v, then it either has the same previous view, or its previous view was disjoint from v (EVSnonoverlap).

The final part of EVS relates views and message ordering. For this automaton, we need to express the relation between the FIFO and VIEW automata, and we include state variables $received_{q,p,v}$ and $all\text{-}views_p$. The VIEW_MSG automaton is shown next.

VIEW_MSG

State: for $p, q \in PID$, $v \in View$:
$received_{p,q,v} \in \mathcal{M}$ List
for $p \in PID$:
$all\text{-}views_p \in View$ Set

EVS-NEWVIEW$_p(v)$
Pre: let $v' = current\text{-}view_p$ in
$\forall q \in v.set$:
$\forall r \in v'.set$. if $pred\text{-}view_{q,v} = v'$ then
$received_{r,p,v'.id} = received_{r,q,v'.id}$

METAPRL

The VIEW_MSG automaton introduces a new precondition for delivering a view: each process q with the same preceeding view v' receives exactly the same messages from each process $r \in v'.set$ (they EVSsync property).

These three automata specify the properties of EVS individually. The final step is combine them into the complete specification of EVS.

$EVS \equiv$ GROUP_FIFO $\cap$ VIEW $\cap$ VIEW_MSG
METAPRL

This construction identifies the $received_{p,q,v}$ variables of the GROUP_FIFO and VIEW_MSG automata, and the $all\text{-}views_p$ variables of the VIEW and VIEW_MSG automata. The properties of EVS are the conjunction of the properties of the three parts, forming a complete specification of EVS. Furthermore, the proofs of the parts are *inherited* by the composition. For example, the proof of the safety property (EVS $\vDash$ *always* EVSvieworder) follows from an application of Theorem 1 to the component proof (EVS_VIEW $\vDash$ *always* EVSvieworder).

4.7 Related Work

Abadi and Lamport [AL95] have explored composition in TLA using *assume-guarantee* specifications. We describe our systems with au-

tomata, rather than a temporal *logic* like TLA, because we can use a single language to represent both programs and specifications. However, our formalism shares the same problem space with TLA. In fact, our safety theorem $(A \vDash \mathit{always}\, P) \Rightarrow (A \cap B \vDash \mathit{always}\, P)$ is currently too weak: while the intersection $A \cap B$ has the safety properties of both A and B, additional safety properties may hold because of interference between the two automata. It is likely that we can use the assume-guarantee style of reasoning to extend our framework with a more complete result.

Although our approaches differ at a high level—TLA is a temporal *logic* and we deal directly with automata, both approaches describe the same problem space.

Automata are used for specifying distributed systems in [Jon94, Fek93]. In [Kar96], protocol layers for point-to-point messaging are formally specified and composed using TLA [Lam94].

I/O automata are defined by Lynch and Tuttle [LT89], and are discussed further in Lynch [Lyn96]. The Nuprl proof development system includes both a logic and a mechanism for reasoning. An early version of the system is described in Constable et. al. [CAB+86]; more recent descriptions can be found in Jackson's thesis [Jac95]. Records are a central part of our formalization; a more complete description of the type theory of records can be found in Hickey [Hic96].

Birman and Joseph presented one of the earliest accounts of virtual synchrony [BJ87]; more recent versions can be found in [Nei96, FLS97, BDGB98, Bir97]. A more complete description of the verification of EVS can be found in [HLvR99].

Part II

The MetaPRL Logical Framework

Chapter 5

The MetaPRL Meta-Language and Logic

The MetaPRL term language and meta-logic provide the framework for all logical constructions in the system. Modules, theories, programs, and logical statements require two basic parts to make relations between statements possible:

1. a common language to express statements and proofs, and
2. a meta-logic to allow relations to be proved between statements in the language.

The meta-logic is a weak constructive logic of second-order Horn formulas, and the meta-language allows the definition of logical inference rules over terms in the meta-language, as well as the definition of computational rewrites (terms that are equivalent in every context). Second-order matching is decidable for terms in the meta-language, and theorem proving uses matching to provide a tactic-tree LCF-style proof methodology.

Higher-order features, such as relations between modules and inheritance of theories, are reduced to expressions in the meta-logic. Tools for proof automation rely on logical definitions in the meta-logic. In this chapter we describe the meta-logic used in MetaPRL in several parts, beginning with the language and meta-logical rules, and working toward the logical module system and proof automation.

5.1 The term language

The term language defines the *syntax* for all definitions, including programs and logical statements. The MetaPRL term language is based

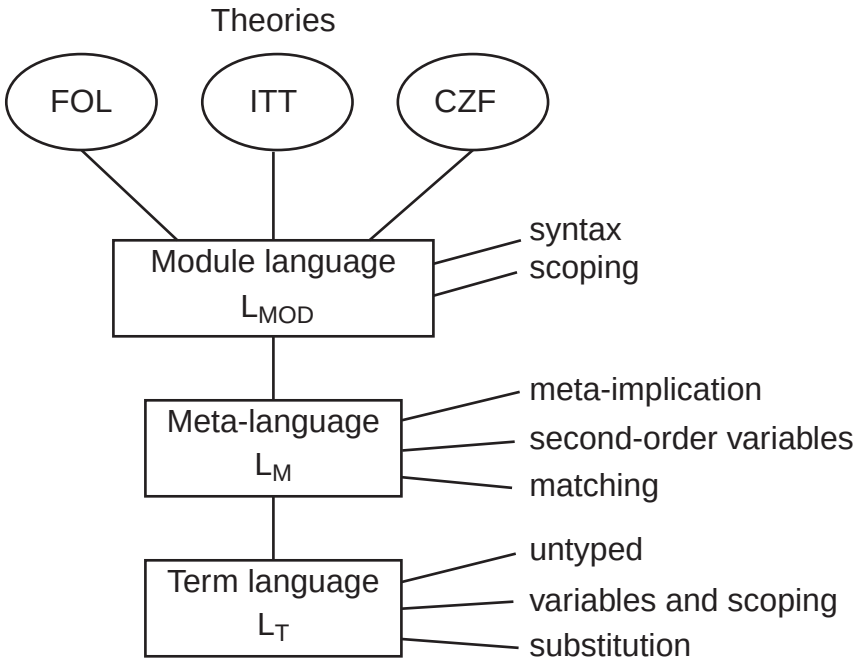

Figure 5.1: Hierarchy of languages

on the Nuprl term language [Jac94]. The term language includes the definition of *operators* to specify atomic formulas, *variables* to specify abstraction, and *scoping rules* that determine the extent of variable bindings. The basic syntactic algorithms, including free-variable computations and substitution are defined at the term level. Since terms are defined before the meta-language, terms are untyped—although logical interpretations may later give a meaning to terms with some terms being interpreted as types and others as programs.

The term language is defined recursively. Let $\mathcal{O}$ be a set of operator names, and let $\mathcal{V} \subseteq \mathcal{O}$ be a set of operators that denote variables. A *bound-term* pairs a term with a set of binding variables. For $t \in$ *term*, and a finite list of variables $\vec{v} \subseteq \mathcal{V}$, the pair $(\vec{v}, t)$ is a *bound-term*. The variables $\vec{v}$ are *bound* in the term t (this will later affect free-variable calculations). A *term* pairs an operator $o \in \mathcal{O}$ with a list of bound terms $\vec{bt} \subseteq$ *bound-term*: $(o, \vec{bt}) \in$ *term*.

This definition of terms is intentionally vague. Formally, we don't specify the set $\mathcal{O}$ of operators, nor do we define the particular set theory used in the term definition. This allows the construction of semantics that use custom interpretations of the terms. As a result, logics are by definition open-ended: Their models include interpretations for extensions of the logic with new terms and new operators.

In the MetaPRL implementation, the operators are defined concretely as a pair of an operator name (a *pathname* in the module system, which is represented as a list of strings) and a list of parameters, where each parameter belongs to a concrete type like the integers or strings.

$$\begin{aligned} \mathit{param} &= \mathbb{Z} \mid \mathit{String} \\ \mathit{operator} &= \mathit{path} \times \overline{\mathit{param}} \end{aligned}$$

The concrete syntax for terms is defined as follows:

term	::=	operator '{' bound-terms '}'
bound-terms	::=	ϵ | bound-term ';' bound-terms
bound-term	::=	$v_1, \ldots, v_n$ '.' term
operator	::=	opname '[' params ']'
params	::=	ϵ | param ',' params
param	::=	*number* ":n" | *string* ":s"
opname	::=	*string* | *string* '!' opname

We give some example terms. At this level, the terms have no predefined interpretation as numbers, functions, etc. These example terms illustrate possible syntax.

The natural numbers may be represented with an operator that has the opname "number", a single parameters the gives the value of the number, and no subterms.

$$\begin{aligned} 0 &\equiv \mathit{number}[0 : n]\{\} \\ 1 &\equiv \mathit{number}[1 : n]\{\} \end{aligned}$$

Variables are a subset of the operators. Like the numbers, we use a single opname for all variables.

$$\begin{aligned} x &\equiv \mathit{variable}["x" : s]\{\} \\ y &\equiv \mathit{variable}["y" : s]\{\} \\ i17 &\equiv \mathit{variable}["i17" : s]\{\} \end{aligned}$$

A lambda-abstraction can be represented as a term with a single bound term of arity one, and an application is a term with two subterms of arity zero.

$$\begin{aligned} \lambda x.b &\equiv \mathit{lambda}[\,]\{x.b\} \\ f(a) &\equiv \mathit{apply}[\,]\{f; a\} \end{aligned}$$

Types and propositions can also be encoded as terms. A simple function type $A \to B$ could be represented by the following term:

$$A \to B \equiv \mathit{fun}[\,]\{A; B\}.$$

Here are some examples of propositions and types with binding variables:

$$\begin{aligned} \forall x \in A.B &\equiv \mathit{all}[\,]\{A; x.B\} \\ \{x\colon T \mid P\} &\equiv \mathit{set}[\,]\{T; x.P\} \end{aligned}$$

For convenience, we will often omit the parameter brackets [] or bound-term lists {} when they are empty.

5.1.1 Syntax definitions

Given the syntax of terms, we can define some standard syntactic concepts. Given a term

$$t = op\{\overline{v_1}.t_1; \cdots; \overline{v_n}.t_n\},$$

the *immediate subterms* of t are the terms $t_1, \ldots, t_n$. The *subterms* of t are the immediate subterms $t_1, \ldots, t_n$ and the subterms of $t_1, \ldots, t_n$.

The variables in $\overline{v_1}, \ldots, \overline{v_n}$ are *binding* variables; the set of binding variables of t are the variables in $\overline{v_1}, \ldots, \overline{v_n}$ as well as the binding variables in its subterms.

A variable v *occurs* in term t if it is a binding variable in t or if it is a subterm of t. The *scope* of a binding variable $v \in \overline{v_i}$ in t is the term t_i. We say that v is *bound* in t_i, and the subterms of t_i that are equal to v are called *bound occurrences* of v. Any occurrence of a variable v' as a subterm of t that is not a bound occurrence is called a *free occurrence* of the variable v' in t. The set of variables that have free occurrences in t are called the *free variables* of t. Term t is *closed* if it has no free variables; it is *open* otherwise.

For any term t, we use the notation $t.operator$ to mean the operator of t, and $t.bound\text{-}terms$ to mean the bound-terms of t. The following algorithm computes the free variables of a term t. Given a vector $\overline{v}$, the function $setof(\overline{v})$ is the set representation of the variables in the vector $\overline{v}$.

Algorithm 1 *Free variables*

$$FV(t) = \bigcup_{(\overline{v}.t') \in t.bound\text{-}terms} FV(t') - setof(\overline{v})$$

Terms are considered to be equivalent with respect to naming of the bound variables. For instance, terms $lambda\{x.x\}$ and $lambda\{y.y\}$ are considered to be equal terms, called *alpha-variants*. In the meta-logic each term t is treated as an alpha-equivalence class, denoting all terms that are alpha-equal to it. The following algorithm determines if two terms belong to the same alpha-equivalence class.

Algorithm 2 *Alpha equivalence Two terms t_1 and t_2 are alpha-equal with respect to a variable function $R \in \mathcal{V} \to \mathcal{V}$, denoted $t_1 =_R t_2$, if the following hold:*

Either

1. *t_1 and t_2 are variables and $R(t_1) = t_2$,*
2. *or all of the following hold:*
   - *$t_1.operator = t_2.operator$,*
   - *t_1 and t_2 have the same number $n \in \mathbb{N}$ of immediate subterms,*
   - *for each bound-term $b_i \in t_1.bound\text{-}terms$ and $b'_i \in t_2.bound\text{-}terms$, b_i and b'_i have the same arity, and if $b_i =$*

$(v_1, \ldots, v_m.t)$ and $b'_i = (v'_1, \ldots, v'_m.t')$, then $t =_{R'} t'$ for the extended variable map

$$R' = R[v_1 \mapsto v'_1][v_2 \mapsto v'_2] \cdots [v_m \mapsto v'_m].$$

Definition 1 *Alpha-equality. Two terms are alpha-equal, denoted $t_1 =_\alpha t_2$ if $t_1 =_R t_2$ for the identity function $R \in \mathcal{V} \to \mathcal{V}$.*

Since alpha-equality uses a renaming function that renames binding occurrences of variables, the free-variable computation should be the same for all terms in a alpha-equivalence class. We prove that result in the following theorem, which uses structural induction on terms. Since terms are finite, they are well-founded. Structural induction on terms is tree induction: to prove a property of P of terms in general, it is sufficient to prove property P for an arbitrary t term assuming property P for the subterms of t.

Theorem 4 *For any two terms t and t', if $t =_R t'$, then $R(FV(t)) = FV(t')$.*

Proof Consider two terms t and t', where $t =_R t'$ for some $R \in \mathcal{V} \to \mathcal{V}$. If t and t' are variables, then

$$R(FV(t)) = R(t) = t' = FV(t').$$

Otherwise, their operators must be equal and they have the same numbers of subterms with the same arities.

$$\begin{aligned} t &= op(\overline{v_1}.t_1; \cdots; \overline{v_n}.t_n) \\ t' &= op(\overline{v'_1}.t'_1; \cdots; \overline{v'_n}.t'_n) \end{aligned}$$

The definition of equality requires that $t_i =_{R'} t'_i$ for $R' = R[v_{i_1} \mapsto v'_{i_1}] \cdots [v_{i_m} \mapsto v'_{i_m}]$. By the induction hypothesis, we have

$$R'(FV(t_i)) = FV(t'_i).$$

If we subtract the binding variables, we get the following equivalence,

$$R(FV(t_i) - setof(\overline{v_i})) = FV(t_i) - setof(\overline{v'_i}).$$

By the free-variable algorithm,

$$R(FV(t)) = R(\bigcup_{(\overline{v_i}.t_i) \in t.bound\text{-}terms} FV(t_i) - setof(\overline{v_i})).$$

The relation R can be commuted with the union, to get

$$R(FV(t)) \quad = \bigcup_{(\overline{v_i}.t_i) \in t.\textit{bound-terms}} R(FV(t_i - \textit{setof}(\overline{v_i})))$$

Corollary 1 *Free-variable computations are functional with respect to alpha-equality: if $t =_\alpha t'$, then $FV(t) = FV(t')$.*

Proof If $t =_\alpha t'$, then $t =_R t'$ for the identity function $R \in \mathcal{V} \to \mathcal{V}$. So $R(FV(t)) = FV(t) = FV(t')$.

The final syntactic operation is *substitution*. The notation $t[v_1 \leftarrow t_1; \cdots; v_n \leftarrow t_n]$ means the simultaneous replacement of the free occurrences of $v_1, \ldots, v_n$ in the term ,with the terms $t_1, \ldots, t_n$. Since terms are treated as alpha-equivalence classes, substitution is allowed to perform renaming to preserve free variable computation. If all variables $v_1, \ldots, v_n$ are free in t, the free variable property we want is the following:

$$FV(t[v_1 \leftarrow t_1; \cdots; v_n \leftarrow t_n]) = (FV(t) - \{v_1, \ldots, v_n\}) \cup (\bigcup_{i \in \{1,\ldots,n\}} FV(t_i)).$$

A substitution algorithm that this property for all substitutions is called a *capture-avoiding* substitution.

Before defining substitution, we specify the property for a *substitution instance*, which is a generalization of the alpha-equivalence property.

Definition 2 *For any two terms t and t', t' is a* substitution instance *of t, written substitution $t =_S t'$ for $S \in \mathcal{V} \to$ term if either:*

1. *t is a variable, and $t =_\alpha S(t)$*
2. *or all of the following hold:*
   - *$t.\textit{operator} = t'.\textit{operator}$,*
   - *t and t' have the same number of immediate subterms with the same arities,*
   - *for each bound term $b_i \in t.\textit{bound-terms}$ and $b'_i \in t'.\textit{bound-terms}$, if $b_i = v_1, \ldots, v_m.t_i$ and $b'_i = v'_1, \ldots, v'_m.t'_i$, then $t_i =_{S'} t'_i$, where $S' = S[v_1 \mapsto v'_1] \cdots [v_m \mapsto v'_m]$.*
   - *$FV(t') = \bigcup_{v \in FV(t)} FV(S(v))$ (the free variable property).*

The key difference between this definition and the definition of alpha-equality is the introduction of case 4, for free variable computations. In order to preserve this property in the algorithm for substitution, we need to rename variables during the substitution to avoid capture. The following algorithm defines the capture avoiding substitution.

Algorithm 3 Capture avoiding substitution. *Let $t \in term$ be a term, and $S \in \mathcal{V} \to term$ be a substitution function. Define the function $V(t) = \bigcup_{v \in FV(t)} FV(S(v))$ to compute free variables in the range of S. To compute a substitution instance $t' \in term$, where $t =_S t'$, perform the following.*

1. *If t is a variable, let $t' = S(t)$.*
2. *Otherwise, let $t = op(\overline{v_1}.t_1; \cdots; \overline{v_n}.t_n)$. Then $t' = op(\overline{v'_1}.t'_1; \cdots; \overline{v'_n}.t'_n)$, where v_{ij} is a* new *variable where $v_{ij} \notin FV(t_i) \cup V(t)$, and $t'_i =_{S'} t_i$ for*

$$S' = S[v_{ij} \mapsto v'_{ij}] \quad for j \in \{1, \ldots, |\overline{v_i}|\}.$$

The finite substitution $t[v_1 \leftarrow t_1; \cdots; v_n \leftarrow t_n]$ is defined to be $t(I[v_i \mapsto t_i]^{i \in \{1,\ldots,n\}})$, where I is the identity function.

As the final result of this section, we show that the substitution algorithm produces a substitution instance.

Theorem 5 *For any substitution $S \in \mathcal{V} \to term$, if $t' = t(S)$ then $t' =_S t$.*

Proof The proof is by structural induction on t. If t is a variable, then $t = S(t) =_S t$. Otherwise, by the definition of substitution,

$$\begin{aligned} t &= op(\overline{v_1}.t_1; \cdots; \overline{v_n}.t_n) \\ t' &= op(\overline{v'_1}.t'_1; \cdots; \overline{v'_n}.t'_n), \end{aligned}$$

where $|\overline{v_i}| = |\overline{v'_i}|$. Furthermore, the variables $\overline{v'_i}$ do not occur free in t_i or $V(t_i)$. By induction

$$t'_i =_{S'} t_i \quad for S' = S[v_{ij} \mapsto v'_{ij}]^{j \in \{1,\ldots,|\overline{v_i}|\}}.$$

The free variable property provides

$$FV(t'_i) = \bigcup_{v \in FV(t_i)} FV(S'(v)).$$

By equational reasoning,

$$\begin{aligned} FV(t') &= \bigcup_{i\in\{1,\ldots,n\}} FV(t'_i) - setof(\overline{v_i}) \\ &= \bigcup_{i\in\{1,\ldots,n\}} \left(\bigcup_{v\in FV(t_i)} FV(S'(v)) \right) - setof(\overline{v_i}). \end{aligned}$$

For each v_{ij}, if $v_{ij} \in FV(t_i)$, then $S'(v_{ij}) = v'_{ij}$, so we can rewrite the union as follows.

$$FV(t') = \bigcup_{i\in\{1,\ldots,n\}} \left(\begin{array}{l} \bigcup_{v\in FV(t_i)\cap setof(\overline{v_i})} FV(S'(v)) \\ \cup \bigcup_{v\in FV(t_i)-setof(\overline{v_i})} FV(S'(v)) - setof(\overline{v'_i}) \end{array} \right)$$

Since the v'_{ij} are not free in $\bigcup_{v\in FV(t_i)}$, we have

$$\begin{aligned} FV(t') &= \bigcup_{i\in\{1,\ldots,n\}} \left(\begin{array}{l} \left(\left(\bigcup_{v\in FV(t_i)\cap setof(\overline{v_i})} FV(S'(v)) \right) - setof(\overline{v'_i}) \right) \\ \cup \left(\left(\bigcup_{v\in FV(t_i)-setof(\overline{v_i})} FV(S'(v)) \right) - setof(\overline{v_i}) \right) \end{array} \right) \\ &= \bigcup_{i\in\{1,\ldots,n\}} \left(\{\} \cup \bigcup_{v\in FV(t_i)-setof(\overline{v'_i})} FV(S(v)) \right) \\ &= \bigcup_{v\in FV(t)} FV(S(v)) \end{aligned}$$

This completes the section on the term language. We have defined several important concepts: free variables, alpha-equivalence of terms, and substitution. We have also shown that the substitution algorithm is correct in the sense that it produces a substitution instance (and avoids capture). The algorithms presented are taken from the ML program code in the implementation, so we have gained some confidence in the term language implementation.

5.2 The frame

In introducing a meta-logic, we must be concerned with the issue of what *concepts* should be part of the meta-logic, and which should be left to the developer. De Bruijn [N.d91] calls the logical framework the

frame; the rules and axioms that the user of the frame defines as the basis for further work are part of the *book*. As de Bruijn explains, there are many reasons (theoretical, practical, educational, and sociological) why the frame should be as simple and as weak as possible. One of the most significance reasons is that *impurities* should be expelled from the framework, making it easier to see the demarkation between the frame and the book, and thus understand the parts.

For our meta-logic, we adhere to de Bruijn's principle, but diverge in the implementation. In his article, de Bruijn used the strongly-typed lambda calculus for the frame; systems like LF [HHP93] and Isabelle [NP92] also use a *typed* logic for the frame. For our purposes, an untyped logic seems more natural, especially since one of the primary *books* we are concerned with is the Nuprl type theory, which uses types to classify untyped programs. In addition to omitting the type system, we also omit *computation* in the form of λ-abstraction and function application. Instead, we use the class of untyped second-order *Horn* formulas for the meta-logic. Horn formulas define *implication*, and second-order quantification provides *substitution*, which together give us the parts we need to formalize both logical consequence and computation.

To explain the logic, we being with *first-order* Horn formulas. In general, a Horn formula can be viewed as an implication. In curried form, a first-order Horn formula of arity n has the form:

$$\forall v_1, \ldots, v_m. t_1 \longrightarrow t_2 \longrightarrow \cdots \longrightarrow t_n,$$

where each of the t_i are formulas, and the v_j are variables. For example, the Horn formula

$$\forall P, Q. (P \wedge Q) \longrightarrow P$$

has two subterms $P \wedge Q$ and P, where P and Q range over formulas, and the clause states that if the formula $P \wedge Q$ is true, then so is P. In general, we will omit universal quanitifers and consider all free variables to be implicitly universally quanitified.

We will call the Horn formulas *meta*-formulas, to distinguish them from normal formulas. A meta-formula of arity zero is a *constant*; meta-formulas of higher arity are *meta-implications*. The truth of a meta-formula depends on the truth of its parts. A constant meta-formula t_1 is true if the formula t_1 is atomically true. A meta-implication $t_1 \longrightarrow \cdots \longrightarrow t_n$ is true if, whenever $t_1, \ldots, t_{n-1}$ are true, then so is t_n.

In MetaPRL the formulas t_i are *terms*, defined in Section 5.1. The *book* classifies the terms that denote valid syntax. Many books, including all of the logics currently implemented in MetaPRL (Part V), define terms for *sequents*, which are used to define the axioms and inference

rules in the book. First-order formulas are not adequate for expressing sequent rules; for that we need *second-order* Horn formulas.

Second-order formulas introduce two new objects: *functions* on formulas, and *applications* of these functions to formulas. Functions and application are defined by introducing second-order variables, called *function symbols*. For example, let a be a second order variable of arity 0, and let $b[\cdot]$ and $C[\cdot]$ be second-order variables of arity 1. The following second-order Horn formula expresses computation by beta-reduction:

$$C[(\lambda v.b[v])\ a] \longrightarrow C[b[a]].$$

In this meta-implication, $b[v]$ represents a formula b with free variable v, and the application $b[a]$ represents the *substitution* of formula a for v in b.

Sequent rules are also expressible in this syntax. For example, the following clause specifies a rule for existential introduction.

$$\begin{array}{l}\Gamma[\vdash a \in A] \longrightarrow \\ \Gamma[\vdash B[a]] \longrightarrow \\ \Gamma[\vdash \exists x{:}A.B[x]]\end{array}$$

In this rule, the sequent has a set of *hypotheses* represented by the second-order variable $\Gamma[\cdot]$, and a single *conclusion* represented by the term after the turnstile. Each sequent is a formula, and $\Gamma[\cdot]$, a, A, and $B[\cdot]$ are second-order variables. The rule is normally applied in backward-chaining style. It's English description is, "The existential $\exists x{:}A.B[x]$ is true in context Γ if there is a witness a of type A where $B[a]$ is true in context Γ." In the MetaPRL implementation, the brackets surrounding the $\vdash$ are normally omitted, and the rule is written as follows.

$$\begin{array}{l}(\Gamma \vdash a \in A) \longrightarrow \\ (\Gamma \vdash B[a]) \longrightarrow \\ (\Gamma \vdash \exists x{:}A.B[x])\end{array}$$

To define the second-order Horn formulas formally, let F^n be the set of function symbols of arity n.

Definition 3 *The second-order terms are the smallest set with the following definition:*

$$\begin{array}{rcl} term_{SO} & = & (\mathcal{O} \times term_{SO}\ list) + App_{SO} \\ app_{SO} & = & \bigcup_{n \in \omega} F^n \times term_{SO}. \end{array}$$

In the concrete syntax provided by the implementation, second-order applications use the notation $\mathcal{F}[t_1; \cdots; t_n]$ for a function symbol

$\mathcal{F}$ of arity n and second-order terms $t_1, \ldots, t_n$. The second-order term language is defined with the following syntax:

$$\begin{array}{rcl} term_{SO} & ::= & \text{operator '\{' bound-terms}_{SO}\text{ '\}'} \\ & | & \mathcal{F}\text{ '[' }terms_{SO}\text{ ']'} \\ bound\text{-}terms_{SO} & ::= & v_1, \ldots, v_n.term_{SO} \end{array}$$

Second-order Horn formulas introduce universal quantification over function symbols, in prenex form.

Definition 4 *A* second-order Horn formula *is a formula with the following shape:*

$$\forall \mathcal{F}_1 \in F^{n_1}. \cdots \forall \mathcal{F}_m \in F^{n_m}.t_1 \longrightarrow \cdots \longrightarrow t_k,$$

where $t_1, \ldots, t_k$ are second-order terms, and $\mathcal{F}_i \in F^{n_i}$ is a function symbol of arity n_i. The scope *of the functions symbols is the clause $t_1 \longrightarrow \cdots \longrightarrow t_k$.*

The definition of free-variables can be extended to second-order clauses.

Definition 5 *If $r \in H$ is a second-order Horn formula, the first-order free variables of r are defined as follows:*

1. *if $r = \forall \mathcal{F} \in F^n.r'$, then $FV(r) = FV(r')$,*
2. *if $r = t \longrightarrow r'$, then $FV(r) = FV(t) \cup FV(r')$,*
3. *if $r = \mathcal{F}[t_1; \ldots; t_n]$, then $FV(r) = \bigcup_{i \in \{1,\ldots,n\}} FV(t_i)$,*
4. *if r is a variable, then $FV(r) = \{r\}$,*
5. *if $r = op(\overline{v_1}.t_1; \cdots; \overline{v_n}.t_n)$, then $FV(r) = \bigcup_{i \in \{1,\ldots,n\}} (FV(t_i) - setof(\overline{v_i}))$.*

The free function symbols are defined as follows.

1. *if $r = \forall \mathcal{F} \in F^n.r'$ then $FV_{SO}(r) = FV_{SO}(r') - \{\mathcal{F}\}$,*
2. *if $r = t \longrightarrow r'$, then $FV_{SO}(r) = FV_{SO}(t) \cup FV_{SO}(r')$,*
3. *if $r = \mathcal{F}[t_1; \ldots; t_n]$, then $FV_{SO}(r) = \{\mathcal{F}\} \cup \bigcup_{i \in \{1,\ldots,n\}} FV_{SO}(t_i)$,*
4. *if r is a variable, then $FV_{SO}(r) = \{\}$,*
5. *if $r = op(\overline{v_1}.t_1; \cdots; \overline{v_n}.t_n)$, then $FV_{SO}(r) = \bigcup_{i \in \{1,\ldots,n\}} FV_{SO}(t_i)$.*

The meta-logic is defined for *closed* second-order Horn formulas H^0 - those formulas $r \in H$ for which $FV(r) = \{\}$ and $FV_{SO} = \{\}$. We typically omit the second order quantifiers in formulas of H^0; function symbols are implicitly universally quantified. For example, in the following formula, the quantification of the constant function symbols A and B is understood:

$$A \longrightarrow B \longrightarrow \mathit{implies}\{A; B\}.$$

5.2.1 The meta-logic

The logic is a deductive system over closed second-order Horn formulas in H^0. Let Σ be a (possibly infinite) set of formulas in H^0, and let σ be a second-order Horn formula. We say that σ is a *consequence* of Σ if the validity of the formula σ follows from the validity of the formulas in Σ. The notion of validity relies on a constructive judgment-as-types interpretation. If

$$r = \forall \mathcal{F}_1 \in F^{n_1}. \cdots \forall \mathcal{F}_m \in F^{n_m}. t_1 \longrightarrow \cdots \longrightarrow t_k$$

is a second-order Horn formula, it is *inhabited* by proof terms that correspond to the proofs of r. Any proof term for the judgment r is a function.

5.2.2 The sequent logic

Since we consider only closed formulas in H^0, we omit quantifiers over function symbols. All formulas are implicitly quantified over all free function symbols. For the remainder of this chapter, lower-case letters $a, b, c, \ldots$ are meta-variables representing second-order formulas. The upper-case letters $A, B, C, \ldots$ represent function symbols, with arity determined by context. The Greek letter Γ represents a set of Horn formulas. The meta-logic has two rules, shown in Table 5.1.

The **axiom** rule states that a goal term is provable from an assumption. All other reasoning uses the *refine* rule which is essentially a limited form of resolution. In this rule, F is function symbol (say with arity k) and $G \in \mathit{term}_{SO}^k \rightarrow \mathit{term}_{SO}$ is a *function* on second-order terms. The second-order term $t'_m[G/F]$ is a "specialization" of t'_m where all second-order applications $F[t''_1; \cdots; t''_k]$ have been replaced by $G(t''_1, \ldots, t''_k)$. The term $t'_m[G/F]$ must be alpha-equal to the head of the goal clause t_n (it must "match"). Note that this may modify the second-order binding structure if the function G introduces new function symbols.

Table 5.1: Rules of the meta-logic

$$\frac{}{\Gamma \vdash t_1 \longrightarrow \cdots \longrightarrow t_n \quad (\text{for } t_n = t_i,\ i \in \{1, \ldots, n-1\})}\ \textit{axiom}$$

$$\frac{\begin{array}{l}1.\ \ (t'_1 \longrightarrow \cdots \longrightarrow t'_m) \in \Gamma \\ 2.\ \ t'_m[G/F] = t_n \\ 3.\ \ \Gamma \vdash t_1 \longrightarrow \cdots \longrightarrow t_{n-1} \longrightarrow t'_i[G/F] \quad \textit{for } i \in \{1, \ldots, m-1\}\end{array}}{\Gamma \vdash t_1 \longrightarrow \cdots \longrightarrow t_n}\ \textit{refine}$$

Example: a small propositional logic

Let's consider a propositional logic. We represent propositions as terms, with the following special terms:

$$\texttt{false} \text{ represents } \bot, \ \texttt{implies}\{t_1; t_2\} \text{ represents } t_1 \Rightarrow t_2.$$

The propositional rules are phrased in a sequent calculus, where a sequent is expressed as a nested term as follows

$$\texttt{hyp}\{t_1; \texttt{hyp}\{t_2; \cdots \texttt{hyp}\{t_n; \texttt{concl}\{t\}\} \cdots\}\}.$$

This term represents the sequent

$$t_1,\ t_2,\ \cdots,\ t_n \vdash t.$$

The calculus has a judgment $\texttt{wf}\{s\}$ for syntactical well-formedness of a sequent s.

For clarity, we print the terms using their normal notation, sometimes mixing it with the concrete syntax.

$$\begin{array}{rcl} P, Q, R, \ldots & \equiv & \texttt{prop["P":s],prop["Q":s],prop["R":s],...} \\ \bot & \equiv & \texttt{false} \\ P \Rightarrow Q & \equiv & \texttt{implies\{P; Q\}} \\ P_1,\ P_2,\ \ldots,\ P_n \vdash P & \equiv & \texttt{hyp\{P1; hyp\{P2;...hyp\{Pn; concl\{P\}\}\}\}} \end{array}$$

The next step in defining this logic is to encode the rules of the logic as meta-implications; these rules are shown in Table 5.2. In the meta-logic, these rule establish the context Γ for proofs in the propositional logic.

This logic uses the two function symbols F and G to represent parts of a sequent, and P, Q, and R are function symbols that represent propositions. The symbol F represents a list of hypotheses of a sequent, and

Table 5.2: Rules of the propositional logic

wf-base	$\texttt{wf\{concl\{}G\texttt{\}\}}$
wf-hyp	$\texttt{wf\{}S\texttt{\}} \longrightarrow \texttt{wf\{hyp\{}H\texttt{;}S\texttt{\}\}}$
axiom	$\texttt{wf\{}F[P\texttt{;}G[\texttt{concl\{}P\texttt{\}}]]\texttt{\}} \longrightarrow F[P\texttt{;}G[\texttt{concl\{}P\texttt{\}}]]$
any	$\texttt{wf\{}F[\texttt{hyp\{false;}G\texttt{\}}]\texttt{\}} \longrightarrow F[\texttt{hyp\{false;}G\texttt{\}}]$
intro	$\texttt{wf\{}F[\vdash P \Rightarrow Q]\texttt{\}} \longrightarrow F[P \vdash Q] \longrightarrow F[\vdash P \Rightarrow Q]$
elim	$\texttt{wf\{}F[P \Rightarrow Q,\ G[\vdash R]]\texttt{\}}$ $\longrightarrow F[P \Rightarrow Q,\ G[\vdash P]]$ $\longrightarrow F[P \Rightarrow Q,\ G[Q \vdash R]]$ $\longrightarrow F[P \Rightarrow Q,\ G[\vdash R]]$

G represents the remaining part of the sequent. Each rule requires well-formedness reasoning to restrict the functions to syntactically correct sequents.[1]

Let's look at a fragment of the proof of the proposition $A \Rightarrow ((A \Rightarrow \bot) \Rightarrow \bot)$. The goal clause is the sequent $\vdash A \Rightarrow ((A \Rightarrow \bot) \Rightarrow \bot)$, which is the term

```
concl{implies{prop["A"];
    implies{implies{prop["A"]; false}; false}}}.
```

Each step in a proof consists of selecting a rule from the context, and *matching* the rule with the goal clause. An initial part of the proof is shown in Table 5.3.

5.3 Related Work

We have mentioned the Isabelle [NP92] and LF [HHP93] logical frameworks. Research in this area initially started with de Bruijn [deB70, deB80, NGV94] in the AUTOMATH project with the purpose of formalizing mathematical arguments that could be machine checked.

The Maude [MOM93, MOM94, CDE+99] system is a very general framework that uses *rewriting logic* as a meta-logic. Maude provides a modular framework that supports relations between modules. At the level of meta-theory, MetaPRL and Maude have a lot in common; the generalized Horn clauses in MetaPRL are quite similar to rewriting

[1] Since sequents are used so often to define logics, the implementation provides syntactical support for automated well-formedness checking on sequents, and these subgoals are typically left out.

Table 5.3: Part of the proof of $A \Rightarrow ((A \Rightarrow \bot) \Rightarrow \bot)$

$\vdash A \Rightarrow ((A \Rightarrow \bot) \Rightarrow \bot)$
BY intro (with
P = prop["A"]
Q = implies{implies{prop["A"]; false}; false}
$F[t]$ = t)
| wf{$\vdash A \Rightarrow ((A \Rightarrow \bot) \Rightarrow \bot)$}
| BY wf-base
| $A \vdash (A \Rightarrow \bot) \Rightarrow \bot$
| BY intro (with P = implies{prop["A"]; false})
| Q = false
| $F[t]$ = hyp{prop["A"];t}
| | wf{$A \vdash (A \Rightarrow \bot) \Rightarrow \bot$}
| | BY ...
| | A, $A \Rightarrow \bot \vdash \bot$
| | BY ...

statements in Maude. Beyond that, it is difficult to compare MetaPRL and Maude on equal terms: MetaPRL has grown out of constructive type theory, and Maude has grown out of order-sorted algebra. The models are different. One model we frequently use to understand constructive type theory is Allen's semantics [All86], which provides a PER model for terms. Derivation is performed in a sequent calculus where sequents are given a interpretation based on pointwise functionality. We rely on correspondences between proofs and programs, and propositions and types. The models used in Maude also rely on equivalence classes of terms, but the models have a category structure where objects are states, proofs are morphisms, and dynamic behavior is deduction. Our type-theoretic models are open-ended; rewrite theories are typically closed. These *applications* lead to implementations that seem quite different. The exact nature of these differences remains elusive.

The KIDS system [Smi85, Smi90, Smi91] uses formal methods to develop high-performance algorithms, with a strategy similar to the methods we are pursuing in MetaPRL. As Smith describes, program development in KIDS goes through a workflow of 1) developing a domain theory, 2) creating a specification, 3) applying tactics to reduce and refine the specification, 4) applying optimizations, data type refinements, and compiling. In the PRL group, we advocate the same methdology, using constructive type theory as a starting ground. The KIDS system, like Maude, uses category theory as a basis for establishing morphisms to

relate systems, specifications, and components.

Basin and Constable [BC93] have proposed the use of constructive type theory as a meta-logical framework. In de Bruijn's terminology, the *frame* acquires a great deal of expressive power, including an expressive definitional equality. This can be attractive from the user's point of view—a great deal of knowledge is built into the frame. However, the implementation can be difficult for the same reason. MetaPRL is faithful to the meta-logical tradition; it provides a minimal frame for implementing constructive type theory, which is then used to provide a meta-logical framework. In fact, even though the MetaPRL meta-logic is the basis for relations between theories, we often view the Nuprl type theory as the common foundation for explaining other logics. If we carry it far enough, "all relations lead to Nuprl."

Nuprl has long used second-order variables to define inference rules. MetaPRL also uses a *capturing* form of second-order variables to define term rewriting. In MetaPRL a rewrite can be interpreted as two symmetric inference rules defined using a context variable. The rewrite $t_1 \longleftrightarrow t_2$ is the same as the two rules $C[t_1] \longrightarrow C[t_2]$ and $C[t_2] \longrightarrow C[t_1]$. The context $C[\cdot]$ is a second-order variable of arity one that *binds* the free variables of its argument. For example, the rewrite for beta-reduction $C[(\lambda x.b[x])\ a] \longrightarrow C[b[a]]$ can be applied to the inner redex of the term $\lambda y.((\lambda z.z)\ y)$ to obtain the term $\lambda y.y$. The corresponding rule definition would not apply, since y is free in the inner redex.

Part III

The MetaPRL System

Chapter 6

The MetaPRL System Architecture

Any framework for automation must have at least two parts: an "engine" that performs the tasks required for automation, and a user interface that provides a means of specifying and guiding the tasks to be performed. In a logical framework, the task is *theorem proving*: given a statements expressed in some logic, show that the statement can be derived from the basic assumptions and axioms of the logic. While this may sound simplistic, it is in fact a quite general methodology. For instance, here are some common tasks:

optimization: given a program, show that there is another program that is functionally equivalent, but is more efficient with respect to computational resources.

verification: given a program and a specification, show that the program satisfies the specification.

derivation (porting): given two logics, show that the true statements of one can be reduced to the true statements of the other (subject to an interpretation).

In a setting that allows the definition of higher-order logics, like in MetaPRL, the task of theorem proving is in general undecidable, and it can also be quite difficult even with interactive guidance. The goal in MetaPRL is to address the problem in three parts.

1. Judgements should be *foundational*—all proofs should be reducible to the fundamental judgments of the logic; and they should be open to *scrutiny* so that proofs may be summarized for convenience, and additional detail may be shown upon request.

2. The mechanism for proof automation should be extrememly general so that proof procedures may be defined for arbitrary logical domains. In this respect, MetaPRL follows the design of

LCF [GMW79] by using a general programming language (ML) for proof procedures, along with the *tactic-tree* mechanism [ACHA90] in Nuprl for summarizing and explaining a proof.

3. Finally, the iterface with the user should be both general and easy to use. It should assist in the construction of proofs as well as their explanation. To this end, MetaPRL includes a proof editr for examining, constructing, and explaining proofs. MetaPRL retains Nuprl's distinction between user *input*, which must be specified unambiguously (in context), and *output*, which may be displayed according to context and user preferences. As with most mathematical notation, ambiguity may be introduced in the displayed notation in order to emphasize features that depend on the goals of the presentation.

There are at least three important *views* of the MetaPRL architecture: the **implementation** view emphasizes the *design* of the system for the purpose of development; the **programming** view emphasizes the use of MetaPRL as an extension to a programming language, with the consequent notions of syntax, compiling, and execution; and the **interactive** view emphasizes the use of MetaPRL as a tool for logical exploration and education, where the architecture is defined by the concepts provided by the user interface. While these different views are not incompatible, they are distinct—for example, the implementation of a single logical concept may be spread over many different components.

We will discuss each of these views in turn, but first it is useful to understand the general design from the point of view of a user. The typical workflow of a user on a logical environment is a cycle with several stages. The first step in formalizing a problem is to design a logic for expressing the problem. For example, if the task is to verify a component in a program, the first step is to define the syntax of the programming language, and then to define its semantics (the *meaning* of the program) so that it is possible to reason about particular program properties like execution behavior or the possible values a program produces, or its interaction with the environment. The next step in the workflow is to explore the formalization by proving *theorems* about general properties of the formalization or about specific objects, specification, or relationships between formal objects.

At any step in this process, previous work, including previously defined logics, may be useful in the development, and the task may include *relating* the new task to the existing body of work. The development includes discovering these relations and making them explicit in the development. These are the steps in providing a new formalization (shown in Figure 6.1).

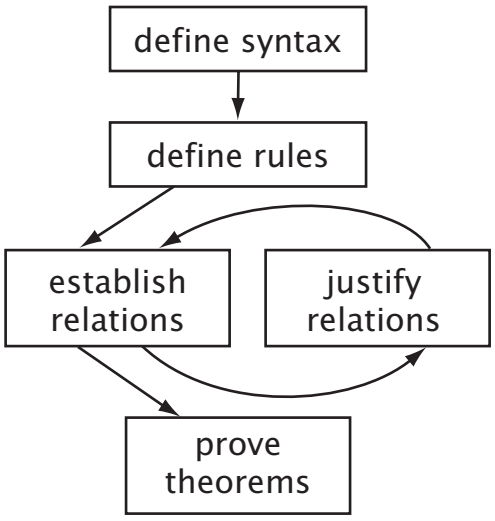

Figure 6.1: Typical workflow

1. Define the *syntax* of the new logic.
2. Define logical *rules* that express the basic definitions and assumptions of the logic.
3. Define any *relations* to existing logics.
4. *Prove* the validiy of the relations.
5. *Augment* the formalization by proving theorems that demonstrate specific properties of the logic.

MetaPRL supports this workflow by organizing formal bodies of work in a *module system*. In fact, the module system is the focal point for all the different views of the system. A MetaPRL module provides primitives for each of the parts of the workflow. The logical foundation for the module system is the meta-logic described in Part II, which provides a uniform syntax for expressing new syntactic definitions and logical rules, as well as expressing the logical content in relations between logics. MetaPRL modules are intended to provide an organizational mechanism for programming as well as mathematics, and in addition to program code, modules can contain any of the following items.

1. Declarations of new syntax.
2. Display definitions that define the printed form of syntax definitions.

3. Relations between syntactic objects that define the *computational equality*.
4. Inference rules that define logical judgments.
5. Declaration of relations between logics, defined by inheritance.
6. Theorems, expressed as inference rules together with proofs, used both for expressing facts and justifying relations between logics.
7. *Tactics* (program code) that implement proof search.

In mathematics, development is often by *extension*. For example, a common method for presenting abstract algebra is to introduce the simpler objects first, starting with a simple object like a monoid, which is defined as a set with a closed, associative, binary operation on the set, and an identity. Once the definitions are in place, the next step is to derive properties of the class of monoids—perhaps to classify particular kinds of monoid according to their properties. These properties are stated as theorems like "all monoids over a set with one element are isomorphic." Once the monoids have been studied sufficiently, the next step might be to introduce *groups*, where a group is a monoid together with an inverse operator. A principle observation is that a group *is a* monoid, and all the theorems of monoids are also theorems about groups. Groups have additional properties, of course, and after these have been developed as additional theorems, the next step might be to introduce *rings* which combine a monoid and a group. Theorems about monoids and groups hold for the corresponding parts in the ring.

The principle behind this method of development is to maximize the re–use of both the presentation and the properties that have been proved—it would be foolish to discard work on monoids when groups are introduced. Modern programming methodology exhibits the same sort of extension property, most obviously with object-oriented programming, where the objective is to re-use code by programming only the *changes* needed to build a new program from the existing programming base. For example, many graphical editors treat documents as a sequence of generic graphical objects, where the "generic" object has generic functions for returning the size of the object and drawing it on the screen (drawing is the responsibility of the component). Generic objects are then extended for particular graphical components: a text field adds a string of characters to be displayed, a rectangle might include a fill and border color. A *colored* text field would be an extension of a generic text field, adding a color and modifying the drawing routine to use the color.

To allow for both kinds of developments, the MetaPRL formal module system includes features for inheritance so that logics (and programs) can be defined by extension. Modules are extended by *adding* additional syntax, rules, and assumptions, and all *theorems* are inherited by module extensions. This support extends to proof automation: each module defines the parts of proof automation relavent in its domain, and these procedures are combined as theories are extended.

MetaPRL maintains a close connection between programming modules and logical modules—in fact, standard programming modules are treated as degenerate logics without any explicit logical content. In normal (non-logical) programming development, program text is entered using a text editor in multiple files, which are then passed to a compiler to be compiled and linked to form an executable. Logical modules augment this by allowing logical statements to be interleaved with program text. Modules containing logical content are passed to the MetaPRL compiler, which contains a theorem prover for checking and compiling the logical content. The result of compilation is a program like any other, where logical assertions have been checked and proof obligations have been fulfilled. Of course, this is somewhat simplified—the theorems defined in a file might not be provable by the compiler. Instead, MetaPRL augments the module editor to include a proof editor where logical statements in a module can be edited and proved when the module is created.

The component architecture is shown in Figure 6.2. The correspondence between this "logical environment" and a more standard programming environment is a matter of enhancement: the editor includes support for editing, browsing, and proving; the logical library plays the part of the filesystem (with enhancements for maintaining consistency and versioning of logical modules), and the compiler is enhanced with tools for proof checking and extraction of computational content from proofs. In the end, the result is a program executable where logical as well as standard program code has been compiled to executable form. The environment is "conservative" in this sense: applications that contain no logical content are compiled by MetaPRL as-is, and the result is in no way different from the result of a standard compilation. However, MetaPRL offers additional features even for non-logical program code, including the abilities of browsing code, tracking dependencies, and debugging based on program *semantics* (rather than testing).

This has been a light overview of the MetaPRL system; the following Chapters provide more detail on parts of the system from each of the points of view menationed previously: the *user*'s view, the *programming* view, and the *implementation* view. Because of its central role in the MetaPRL system, the module system is discussed first (in Chapter 7).

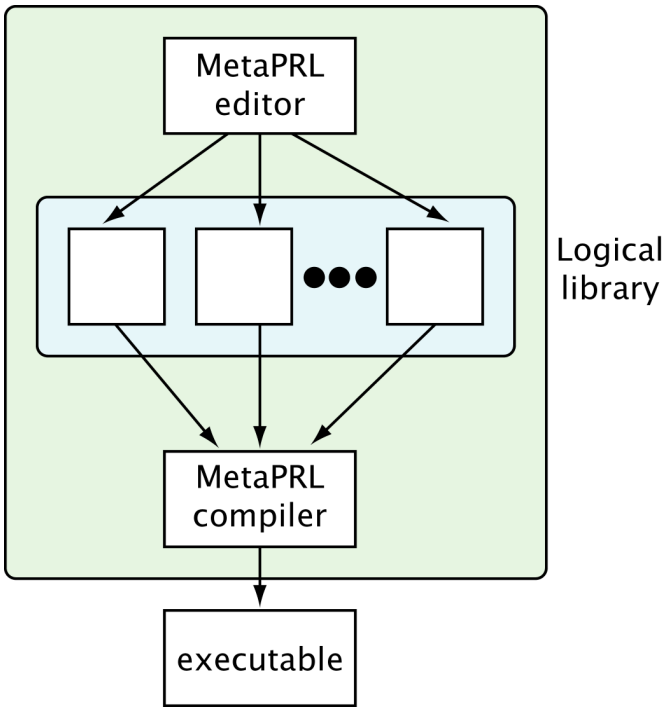

Figure 6.2: Component architecture

The remaining chapters are devoted to the implementation.

Chapter 7

The MetaPRL Module System

One of the goals in MetaPRL is to maintain a close connection between the formal module system and the OCaml programming language. The goal is to make the MetaPRL module system a conservative extension of the OCaml module system. By making the formal system a strict enhancement of OCaml, we provide a path for adding formal reasoning to applications that were previously developed using standard software engineering methodology. This eases the burden of programming in a formal system because formal tools (for specification, verification, documentation, etc.) need only be learned when the benefits of doing so are desired. The reason is also pedantic: to learn how to program in a formal system, we can first learn how to program informally and then augment our knowledge with a foundational mathematical understanding. The final reason is a matter of bootstrapping: we would like to use MetaPRL to reason about its own implementation—but we need an implementation first!

To understand this Chapter about the module system, it is not necessary to understand OCaml. However, we need some basic understanding of what a module system provides. There are two kinds of objects in a module system: **signatures** describe (declare) the components of a module, and **structures** provide the corresponding implementation. Each component has a *name* and a *type*. A structure *type-checks* against a signature if it implements each of the components declared in the signature with a program of the declared type; a structure may have multiple signatures. A **functor** is just a function (its name comes from category theory) on structures.

In a programming language, a structure contains values like functions, numbers, or elements of some abstract data type. A structure may also contain *type declarations* that define a new type with a new name. In the formal system, these values are extended to include declarations of new syntax, proof procedures, and *proofs*. Using the judgments-as-

types principles, the *type* for a proof (used to declare the proof in a signature) is the statement of a theorem that the proof proves.

Modules are interpreted formally in the MetaPRL meta-logic, discussed in detail in Part II. In this Chapter, we'll first give a brief overview of the meta-logic, then we'll cover the new logical components, and then we'll give a complete example of defining a logic.

7.1 The MetaPRL module system

The syntax of formal modules is shown in Table 7.1. The module primitives can be grouped into several categories: the `declare` statement defines new *input* syntax, the `dform` statement defines new output syntax, and the `prec` statement defines precedence levels for parenthesization during printing. The `axiom` statement defines a new inference rule, and the `prim`, `thm`, and `interactive` statements provide justifications and proofs for `axiom` declarations. Rewrite definitions (defined with the `rewrite` statement) are a special form of inference rule that defines the computational equaivalence; the `prim_rw`, `thm_rw`, and `interactive_rw` statements provide justifications and proofs. The `include` statement is used to define module extensions; they specify the inheritance hierarchy on formal modules. The `resource` statement defines an object useful for defining program content by inheritance. In the following sections, we will describe each of these statements in more detail.

7.2 Term syntax and the `declare` statement

Terms are the logical *syntax*. All logical statements are expressed as terms, and new term syntax is introduced with the `declare` directive. All operations in the system are defined on terms: all formal programs are interpreted as terms, and all logical statements in the system are defined with terms. Terms in the MetaPRL implementation have the concrete uniform syntax shown in Table 7.2. This language is modified slightly from the more abstract syntax in Section 5.1; it includes additional parameters types and qualified operator names.

A *simple* literal is any sequence of alphabetic and numeric characters beginning with a letter. Additional literals are defined as any sequence of characters from the ASCII character set enclosed in double quotes. A *string* is any sequence of characters enclosed in double-quotes, and a *number* is any sequence of digits. A *var* is a simple literal. The *bterms*

Table 7.1: Syntax of formal signatures

signature	::=	`sig` *sig_stmt** `end`
sig_stmt	::=	`declare` *term*
	|	`dform` *name* [*df_options*] :: *term*
	|	`prec` *name*
	|	`axiom` *name* p_1 ... p_n : *judgment*
	|	`rewrite` *name* : $t_1 \longleftrightarrow t_2$
	|	`include` *module_name*
	|	`resource` (*type*, *type*, *type*, *type*) *name*
	|	*ML declaration*
structure	::=	`struct` *str_stmt** `end`
thy_stmt	::=	`declare` *term*
	|	`dform` *name* [*df_options*] :: *term* = *df_definition*
	|	`prec` *name*
	|	`prec` $name_1 = name_2$
	|	`prec` $name_1 < name_2$
	|	`axiom` *name* $p_1 \ldots p_n$: *judgment*
	|	`prim` *name* $p_0 \ldots p_n$: *judgment* = *term*
	|	`thm` *name* $p_0 \ldots p_n$: *judgment* = *tactic*
	|	`interactive` *name* $p_0 \ldots p_n$: *judgment*
	|	`prim_rw` *name* : $t_1 \longleftrightarrow t_2$
	|	`thm_rw` *name* : $t_1 \longleftrightarrow t_2$ = *tactic*
	|	`interactive_rw` *name* : $t_1 \longleftrightarrow t_2$
	|	`include` *name*
	|	`resource` (*type*, *type*, *type*, *type*) *name*
	|	*ML implementation*

Table 7.2: Term syntax

term	::=	*operator* { *bterms* }
bterm	::=	*term* | *vars* . *term*
operator	::=	*opname* [*params*]
opname	::=	literal | literal ! *opname*
param	::=	number `:n` | string `:s` | string `:t` string `:v` | *level* `:l`
level	::=	number | literal | *level* number | *level* ' *level* "|" *level*

and *params* lists use the semicolon (`;`) as a separator character, and the *vars* list uses a comma (`,`) as a separator character.

The *operator* represents the *name* of a term, which includes any constant values defined by parameters. The *opname* is the name, the *params* are the constant values, and the *bterms* are the (possibly bound) subterms. If either of the *bterms* and *params* list are empty in a term, the brackets may be omitted.

Variables are a built-in term with the opname `variable`. There are two types of variables. *First-order* variables are defined with the syntax `variable[`*string* `:v]`. *Second-order* variables represent terms with substitutions (second order variables are used to define computational rewrites and inference rules). The syntax of second-order variables represents substitutions with subterms. For example, the term

```
variable["v":v]{variable["v1":v]; variable["v2":v]}
```

is a second-order variable with two variables `v1` and `v2` as subterms.

The MetaPRL parser includes a simplifying syntax for variables: if a simple literal is prefixed with an apostrophe (`'`), the literal is interpreted as a variable. For example, the variable v is also represented with the syntax `'v`, which is expanded by the parser into the full form `variable["v":v]`. Second order variables include their subterms in square brackets, so `'v['v1; 'v2]` is the short form for the second-order variable in the last paragraph.

Term syntax is added to a module with the `declare` directive. The *meaning* of the term is not defined at the time of declaration; the meaning is defined with rewrites and inference rules. Terms can represent constants, like the following constants defined for numbers and strings.

```
declare number[i:n]
declare token[s:t]
```

These declarations use *meta-syntax* for the parameters: the natural number includes a *number* parameter, and the token contains a *token* parameter. Once the meta-syntax has been defined, terms can be included in ML programs as quotations enclosed in CamlP4-style [dR] << · >> quotes:

```
let one = << number[1:n] >>
let hello = << token["hello":t] >>
```

Level-expressions are included in the parameter syntax to define type universes, useful in type-theories like Nuprl to specify a hierarchy of types to classify the types themselves. For example, in the Nuprl type theory module, type universes are declared as

```
declare univ[i:l].
```

Specific instances of type universes are the *base* universe `univ[1:l]`, an *arbitrary* universe with index *i* `univ[i:l]`, and relative universes like `univ[i':l]` (which is the type universe containing `univ[i:l]`). Level expressions can contain multiple indices: the universe `univ[(i | j')':l]` is the minimum type universe containing both `univ[i:l]` and `univ[j':l]`.

Expressions are also defined with the term syntax. For example, the Nuprl module `Itt_int` declares terms to represent arithmetic, including the following:

```
declare add{'x; 'y}
declare sub{'x; 'y}
```

For these terms, the *opname* describes the arithmetic operator, and the subterms represent numbers. The logical syntax is *untyped*, the following terms are valid *syntax*, although the second term has no meaning.

```
let two = << add{1; 1} >>
let error = << sub{3; token["hello":s]} >>
```

Expressions may also include binding. The Nuprl `Itt_rfun` module defines syntax for function abstraction and application.

```
declare lambda{x.  'b['x]}
declare apply{'f; 'a}
```

Binding variables are simple literals (so they are not quoted). The meta-syntax for `lambda` includes a second order variable, indicating that the variable `'x` is *bound* in the body `'b['x]` of the function. The following function is an example of valid syntax in the `Itt_rfun` module:

```
let four = << apply{lambda{x.  add{'x; 'x}}; 2} >>.
```

This just syntax at this point: term evaluation has not yet been defined.

Types are also declared as terms. The syntax does not differentiate between types and programs; the *meaning* of terms is defined by computational rewriting and inference rules. The following are some of the types defined in the `Itt_logic` module:

```
declare true
declare false
declare and{'A; 'B}
declare all{'A; x.  'B['x]}
```

The uniform syntax is verbose: each term contains an operator, and brackets for subterms. The verbosity can obscure the meaning implied by term expressions. The parser also provides a simplified syntax for commonly occurring terms. The following tables list the simplified syntax in the left column, and the right column gives term in the uniform syntax. The simplified form is just an abbreviation; the term must still be declared. For example, the use of the term `true or false` requires that the term for `or` be declared, as in the following sequence:

```
declare true
declare false
declare or{'A; 'B}
let t = << true or false >>
```

7.3 Display forms

The uniform syntax is rather awkward—even with the simplified version of the syntax, the representation can obscure the intended meaning of the formal terms. MetaPRL provides built-in support for pretty-printing logical expressions and OCaml programs. The mechanism is based on *display forms* that use the term rewriter to define programs for term display. The display form mechanism was initially developed by Bates

Table 7.3: Simplified syntax for logical operators

Logical operators	
t_1 `and` t_2	`and`$\{t_1$`;` $t_2\}$
t_1 `&` t_2	`and`$\{t_1$`;` $t_2\}$
t_1 `/\` t_2	`and`$\{t_1$`;` $t_2\}$
t_1 `or` t_2	`or`$\{t_1$`;` $t_2\}$
t_1 `\/` t_2	`or`$\{t_1$`;` $t_2\}$
t_1 `=>` t_2	`implies`$\{t_1$`;` $t_2\}$
`all v:` t_1`.` t_2	`all`$\{t_1$`; v.` $t_2\}$
`exst v:` t_1`.` t_2	`exists`$\{t_1$`; v.` $t_2\}$

Type operators	
t_1 `+` t_2	`union`$\{t_1$`;` $t_2\}$
t_1 `*` t_2	`prod`$\{t_1$`;` $t_2\}$
t_1 `->` t_2	`fun`$\{t_1$`;` $t_2\}$
`v:`t_1 `*` t_2	`prod`$\{t_1$`; v.` $t_2\}$
`v:`t_1 `->` t_2	`fun`$\{t_1$`; v.` $t_2\}$
`x, y:` t_1 `//` t_2	`quot`$\{t_1$`; x, y.` $t_2\}$
t_1 `=` t_2 `in` t_3	`equal`$\{t_3$`;` t_1`;` $t_2\}$
t_1 `!=` t_2 `in` t_3	`nequal`$\{t_3$`;` t_1`;` $t_2\}$
t_1 `IN` t_2	`equal`$\{t_2$`;` t_1`;` $t_1\}$

Arithmetic syntax	
t_1 `+@` t_2	`add`$\{t_1$`;` $t_2\}$
t_1 `-@` t_2	`sub`$\{t_1$`;` $t_2\}$
t_1 `*@` t_2	`mult`$\{t_1$`;` $t_2\}$
t_1 `/@` t_2	`div`$\{t_1$`;` $t_2\}$
t_1 `%@` t_2	`mod`$\{t_1$`;` $t_2\}$
t_1 `<` t_2	`lt`$\{t_1$`;` $t_2\}$
t_1 `<=` t_2	`le`$\{t_1$`;` $t_2\}$
t_1 `>=` t_2	`ge`$\{t_1$`;` $t_2\}$
t_1 `>` t_2	`gt`$\{t_1$`;` $t_2\}$

for Nuprl1 [CAB+86]. Stuart Allen and Richard Eaton made significant improvements [AE00] for the Nuprl4 system [Jac94], in which display forms are also used provide a structured editing mechanism. Allen and Eaton have since developed a powerful general-purpose editing mechanism in Nuprl5.

By comparison, MetaPRL's display mechanism is releatively simple, and it is limited to output formatting. The pretty-printer provides automatic introduction of parentheses based on precedences of display forms, it provides control over indentation and line breaking, and it provides a special character set for display of logical operators.

A display form is defined with the `dform` directive. The syntax of a display form has the following form:

dform *name* [*df_options*] `::` *term* `=` *df_definition*`.`

The *term* part of a display form is a term expressed in the uniform syntax; the term must be defined with the `declare` directive. The *df_definition* is another term that is the display form expansion. Term display is a process of *evaluation*: a term is displayed by selecting the closest-matching display form, replacing it with its expansion, and recursively displaying the result. Naturally, there is a condition for termination, and the definition uses a syntax that includes terms representing strings, line breaking information, and other control terms, as well as recursive display. The options provide information about precedence and parenthesization.

Display form precedences are declared with three additional forms. The first statement, a *declaration* of a precedence level, can occur in both signatures and structures. The second two statements establish relations between precedence levels, and they can occur only within a structure.

prec *name*
prec *name* `<` *name*
prec *name* `=` *name*

Display is part of the logical context of a module, and different modules can define different display mechanisms. The display forms definitions in Table 7.4 illustrate some of the basic display form mechanism for terms in the Nuprl `Itt_logic` module. These definitions produce the following display for the (uniform) term `"and"{true; ."or"{false; true}}` (typed into the MetaPRL toplevel):

```
# << "and"{."true"; ."or"{."true"; ."false"}} >>;;
- :  Refiner.Refiner.Term.term = True ∧ (True ∨ False)
```

Table 7.4: Display forms for a fragment of first-order logic

```
include Nuprl_font
prec prec_and
prec prec_or
prec prec_or < prec_and

dform true_df :  true = ‘"True"
dform false_df :  false = ‘"False"
dform and_df :  parens ::
    "prec"[prec_and] ::  and{'t1; 't2} =
    slot["le"]{'t1} ‘" " wedge ‘" " slot["lt"]{'t2}

dform or_df :  parens ::
    "prec"[prec_or] ::  or{'t1; 't2} =
    slot["le"]{'t1} ‘" " vee ‘" " slot["lt"]{'t2}
```

This module uses the display definitions in the `Nuprl_font` module, used to define the display of `wedge` as "$\wedge$" and `vee` as "$\vee$." It also establishes display precedences to control parenthesization, with the precedence of `or` being less than the precedence of `and`. The display form for `and` has two options: the `parens` option declares that parentheses should be used to enclose the display if the term context has a higher precedence, and the `prec["prec_and"]` specifies the precedence of the display form. The term `and{'t1; 't2}` gives the pattern for terms defined with this display.

The *df_definition* provides the expansion. Display is usually defined as a sequence of terms for each part of the display. The `slot` term is used to parenthesize recursive display, so that the `slot{'t1}` in the `and_df` display form places parentheses around the `'t1` subterm if the precedence is less that `prec_and`. The string ‘" " is a literal string that is printed verbatim; in this case a space character.

The `wedge` term is another recursive display. The display form for `wedge` is defined in the `Nuprl_font` module; its definition is the literal string ‘"\203", which corresponds to the "wedge" character in the Nuprl font.

```
dform wedge_df :  wedge = ‘"\203"
```

7.3.1 Options

Display form options are used to provide precedence information. There are three options:

1. `parens`
2. `prec[`*prec*`]`
3. `inherit`

The `parens` option specifies that the display may be parenthesized. No parentheses are ever added to the display form if this option is not included. The `prec[`*prec*`]` option specifies the precedence to be used in the precedence computation. The `inherit` option is an alternative to `prec[`*prec*`]`: it specifies that the precedence should be inherited from the context. The `inherit` option is used for creating a *transparent* display form, such as the `slot` term, which passes the precedence of its context to its subterm.

7.3.2 The `slot` expression

The `slot` performs the precedence computation for parenthesizing subterms. The general `slot` syntax includes an optional parameter:

`slot[`*option*`]`{*subterm*}

The *option* is a string parameter with two possible values: `slot["le"]`{*t*} specifies that subterm *t* should be parenthesized if its precedence is less than or equal to the precedence of the enclosing term. The `slot["lt"]`{*t*} term specifies that the term *t* should be parenthesized only if its precedence is strictly less than the precedence of its superterm. If the `slot` option is not given, the value defaults to "lt".

There is also a `slot` form for displaying parameters. If the `slot` body is empty, the parameter is displayed using a "standard" display:

- `slot["`*s*`"]` displays string *s* without quotes,
- `slot[`*p*`]` coerces the parameter *p* to a string and displays it without quotes,
- `slot[`*l*`:l]` displays the level expression *l* in normalized form.

7.3.3 Literals

The `'"`*s*`"` form displays the string *s*. It is equivalent to `slot["`*s*`"]`.

7.3.4 Line breaking and zones

The display mechanism divides the output into *zones* that specify line breaking behavior. There are two types of line breaks:

- *Soft* breaks are optional. They are taken only if the page margins would be exceeded without line breaking.
- *Hard* breaks are specified by the zone. By default hard breaks are always taken.

There are three types of zones.

- No line breaking is allowed in *linear* zones. Soft and hard breaks are never taken in linear zones.
- A *soft* zone defines behavior of *hard* breaks; soft breaks remain optional. In a soft zone, hard breaks are not taken if the display in the zone fits on a single line. Otherwise, all hard breaks in the zone are taken.
- A *hard* zone also defines behavior for hard breaks: **all** hard breaks are taken in a hard zone; soft breaks are still optional.

Zones are defined with the terms `lzone`, `szone`, `hzone`, and `ezone`. Zones can be nested; when a zone is terminated (by `ezone`), the enclosing zone is resumed. The `lzone` term begins a linear zone, the `szone` a soft zone, the `hzone` a hard zone, and the `ezone` term terminates the most recently entered zone, and returns to the previous zone.

Line breaking is specified with the terms `sbreak[`*inline*`:s,`*take*`:s]` and `hbreak[`*inline*`:s,`*take*`:s]`. The *inline* string specifies a string to be displayed if the break is *not* taken, and the *take* string is printed before the end of the line if the line break *is* taken. The string arguments can be omitted; they default to the empty string "". For convenience, the `hspace` term is defined in the `Nuprl_font` module as `hbreak[" ":s,"":s]`.

7.3.5 Indentation

Indentation is specified with the left-margin operators `pushm[`*tab*`:n]` and `popm`. These operations establish the location of the left margin. They can be nested; when the left margin is removed with the term `popm`, the left margin is reset with its previous value. The `pushm[`*tab*`:n]` term pushes a new left margin at the current display position plus *tab* characters. The *tab* option can be omitted; it defaults to 0. The `popm` term removes the margin installed with the last `pushm`.

Table 7.5: A complex display form

```
dform ifthenelse_df :  parens ::  "prec"[prec_or]
    ::  ifthenelse{'e1; 'e2; 'e3} =
    pushm[0] szone pushm[3]
    `"if" `" " slot'e1 `" " `"then" hspace
    szone slot'e2 ezone popm hspace
    pushm[3] `"else" hspace
    szone slot'e3 ezone ezone popm popm
```

(At toplevel)

```
# << ifthenelse{."true"; 1; 2} >>;;
- : Refiner.Refiner.Term.term = if True then 1 else 2

# << ifthenelse{gt{ifthenelse{gt{'x; 'y}; 'x; 'y}; 0};
    'x_or_y_is_positive;
    'x_and_y_are_nonpositive};;
- : Refiner.Refiner.Term.term =
if (if (x > y) then x else y) > 0 then
    x_or_y_is_positive
else
    x_and_y_are_nonpositive
```

7.3.6 Example

As an example, we give the display form for the if-then-else term defined in the Nuprl module `Itt_bool`.

This is a fairly complicated display form. The options specify that the `ifthenelse` term may be parenthesized, and it has precedence `prec_or`. The display immediately pushes the current left margin, and enters a soft zone so that the `ifthenelse` term is either displayed on a single line, or all display breaks are taken. The `pushm[3]` term pushes the left margin again to indent the body of the `ifthenelse`.

The condition is displayed on the first line, followed by a possible line break. The body of the conditional is displayed in an additional soft zone. After the body is displayed, the margin is popped so that a line break will return to the margin for the conditional. The *else* string is printed and the left margin is pushed again for the body of the else. Once the else-body is displayed, both soft zones are terminated, and both margins are popped, returning the display state to its original state.

7.4 MetaPRL Axioms and Inference Rules

The MetaPRL concrete syntax for inference rules (the meta-implications defined in Chapter 5) is defined as follows:

judgment ::= *term* | *term* –> *inference*

A *logic* is defined by stating its axioms and inference rules. An *axiom* is an inference rule that is just a single term (axioms don't use the meta-implication). The logics are modular: each module defines a logic that contains the rules defined in the module, and in all ancestor modules.

7.4.1 Signatures

A module signature includes declarations for each of the public axioms and inference rules in a logic. Rules are not *required* to be listed in an interface, but for clarity, many of the logics in MetaPRL, including the Nuprl type theory `Itt_logic`, include the inference rule declarations in their signatures.

Axioms and inference rules are declared with the `rule` form. The general syntax is the following:

rule *name* [*params*] : *judgment*.

The *params* are extra parameters that are passed to the inference rule, including *context* addresses, *term* arguments, and new variable names. The `Itt_struct` module contains a basic axiom:

```
rule hypothesis 'H 'J 'x :
    sequent ['ext] { 'H; x:  'A; 'J['x] >- 'A }.
```

The name of the axiom is `hypothesis`. The parameters include two *context* identifiers `'H` and `'J`, and a variable identifier `'x`. The inference of this axiom defines a set of sequents, containing a sequence of hypotheses defined by the context `'H`, a single hypothesis `x` of type `'A`, additional hypotheses defined by the context `'J['x]` where the variable `x` may occur free, and a single conclusion `'A`. The sequent also contains an extra parameter `'ext` that defines the computational status in the then Nuprl type theory. The parameter is the term `squash` if the axiom has no computational value, and it is a variable otherwise (in this case the variable `'ext`).

The `hypothesis` rule defines a set of valid inferences defined by the terms that is matches. Put another way, the inference states that all substitution instances are valid formulas, where the contexts `'H` and

`'J['x]` are replaced by lists of hypotheses, the variable `'A` is replaced by a term, and the variable `'x` is replaced by another variable. For instance, the following formulas are valid in the `Itt_struct` module:

```
sequent ['ext] { i:  int; z:  false >- false }
sequent [squash] { z:  1; y:  'z + 1 >- 1 }
```

The second example is strange because `1` is not a type. However, the semantics of sequents in the Nuprl type theory validates the truth of the expression. A hypothesis `(x:'A)` in the type theory implies two assumptions: that `'A` is a type, and that `'A` is true. If `'A` is not a type, the assumption is false, so the sequent is trivially true.

Inference rules are also defined using the `rule` directive. An inference is a conditional rule; it states assumptions that must be proved in order for the conclusion to hold. Most introduction and elimination rules have this form. For example, in the `Itt_logic` module, the introduction rule for logical conjunction is stated as follows:

```
rule and_intro 'H :
    sequent ['ext] { 'H >- 'a1 } -->
    sequent ['ext] { 'H >- 'a2 } -->
    sequent ['ext] { 'H >- "and"{'a1; 'a2} }
```

The `'H` is the context identifier used to instantiate the hypotheses, and the `'a1` and `'a2` terms are the subterms of the conjunction. The rule states that in order to prove the conjunction `'a1 & 'a2` in any context `'H`, it is sufficient to prove `'a1` and `'a2` in the same context.

7.4.2 Implementation (structures)

Axioms and inference rules are implemented two ways: they can be derived from previous rules, are they can be *primitive* axioms of a logic. The difference between these two forms is in the way the system does logical accounting.

A primitive inference is "justified" by giving a relation on proofs with the `prim` form, which has the following syntax:

prim *name* `[`*params*`]` `:` *bound_judgment* `=` *term*

A *bound_judgment* is like a *judgment*, but it contains bindings for the proofs. The bindings are bound in the *term* part of the `prim` form, which provides the final proof term. In constructive logics, the "proof" is usually the proof *extract*: a term that defines the computational content of the proof. For example, the following definition gives an implementation of the `and_intro` rule declared above:

```
prim and_intro 'H :
    ('t1 :  sequent ['ext] { 'H >- 'a1 }) -->
    ('t2 :  sequent ['ext] { 'H >- 'a2 }) -->
    sequent ['ext] { 'H >- "and"{'a1; 'a2} } =
    pair{'t1; 't2}
```

The `pair` term is the proof term of the proof: if `'t1` is a proof term for `'a1` and `'t2` is a proof term for `'a2` (both in context `'H`), then `pair{'t1; 't2}` is a proof term for `'a1 & 'a2`. The proof term is especially useful in constructive logics; in classical logics the proof term may just be a canonical proof term, indicating the truth of the proof statement without any computational value.

The system automatically constructs proof terms for complete proofs, using the constructions defined by the `prim` form in the logic. Inferences can also be *derived* from previous rules in a logic. Derived rules are delcared with the following form:

interactive *name* `:` `[`*params*`]` `:` *judgment*

The `interactive` form has exactly the same form as an rule, and it introduces a proof obligation to be satisfied later using the proof editor.

7.4.3 Tactics

When a rule is declared, the system creates a *tactic* that can be used to apply the rule. Tactics are the general reasoning mechanism used for interactive proving. Tactics are OCaml functions, and they can be combined with the *tacticals* in the system description to provide automated proving. The exact type of the tactic depends on the types of its parameters. A context parameter requires an *address* argument, a variable parameter requires a *string*, and a term argument requires a *term*. For instance, the OCaml definitions provided by the rules above are the following:

```
val hypothesis :  address -> address -> string -> tactic
val and_intro :  address -> tactic
```

7.5 MetaPRL computational rewriting

MetaPRL defines an additional kind of logical inference: *computational rewriting*. The history of the MetaPRL implementation can be derived from on Paulson's work in Cambridge LCF [Pau83], in which *conversions* and *conversionals* functions as rewriting "tactics." Jackson

used a similar methdology in Nuprl4 [Jac94] to implement a general-purpose rewriting system that used the *conversion* mechanism to perform computation and general-purpose simplification. In the Isabelle system [NP92], Paulson included a general mechanism for simplification using rewrite rules to build simplification procedures. The MetaPRL mechanism retains the style used in LCF and Nuprl4, but extends the rewriter to include conversions based on *derived* rewrite rules.

Rewrites are declared with the following syntax.

rewrite *name* [*params*] : [*conditions*]
redex_term <--> *contractum_term*

A simple rewrite is defined as an equivalence between terms. For instance, the rewrite for beta-reduction is defined in the `Itt_rfun` module as follows:

```
rewrite reduceBeta : (lambda{v. 'b['v]} 'a) <--> 'b['a]
```

This rewrite pattern defines the beta redex and its contractum, which is formed by substituion. When the rewrite is applied to a beta-redex, the term matched by `'a` is substituted for the variable matched by `'v` in the body `'b`.

Rewrites can also be conditional: they may *depend* on conditions provable in the context. Conditional rewrites are only defined for sequent calculi. Conditions are specified with the meta-implication `-->`. The following rewrite is an example of conditional rewriting in the `Itt_int` module:

```
rewrite unit : ('i != 0 in int) --> (('i /@ 'i) <--> 1)
```

Parameters provide extra variables and terms that are needed for rewriting. For example, a rewrite for reversing the beta reduction requires a term specifying the abstracted body (since it is not unique), a variable representing the bound variable, and also the argument of the application.

```
rewrite inverseBeta lambda{v. 'b['v]} 'a :
    'b['a] <--> (lambda{v. 'b['v]} 'a)
```

7.5.1 Implementations

Rewrites are implemented with two forms: rewrites that are *primitive* to a logic use the `prim_rw` form:

prim_rw *name* [*params*] : [*conditions*] *redex* <--> *contractum*

If a declaration is given for the rewrite, the form in the `prim_rw` must be the same (with possible renaming in the bound variables).

Rewrites that are *derived* from previous rules in the logic are implemented with the `interactive_rw` form. The definition has the same form as the `prim_rw`:

interactive_rw *name* [*params*] : [*conditions*]
redex <--> *contractum*

The `interactive_rw` form establishes a proof obligation, which may be fulfilled with the proof editor. The rewrite may be *used* in subsequent proofs whether it has been verified or not. When a logical theory is finished, the editor can be used to check for unsatisfied subproofs. Primitive rewrites never require a proof obligation, unsatisfied interactive rewrites generate an error during proof checking.

7.6 Rule definitions and proofs

`axiom, prim, thm` A signature declares the rules and structure of a theory, without giving it an implementation, and a theory implementation provides proofs for the declarations. A rule is declared in a theory with an `axiom` form. The declaration

$$\texttt{axiom}\ \mathit{name}\ p_1 \ldots p_n : t_1 \longrightarrow \cdots \longrightarrow t_m$$

specifies that the term t_m is true, if the antecedents $t_1, \ldots, t_{m-1}$ are true. This axiom declares the inference rule, but it does not justify it. To justify an `axiom`, the implementation may specify that the axiom is true by *assumption*, with the `prim` form, or it may derive it from other rules as a `thm`. The `prim` form is used to implement the rules that are primitive in the type theory, and for constructive logics, it also specifies the proof extract term. For instance, the and-introduction rule might be "implemented" as follows.

$$\texttt{prim}\ \mathit{and_elim} : (a{:}\Gamma \vdash A)\ (b{:}\Gamma \vdash B) = (\langle a, b\rangle : \Gamma \vdash A \wedge B)$$

The `prim` form is used only when the type theory is defined. Afterwards, rules are justified as theorems, using the `thm` form, which provides a *tactic* to prove the specified goal. A *tactic* is a packaged rule for backward chaining that includes a method for finding the justification of a rule from an implementation. When an `axiom` is declared, the framework generates a *tactic* that is used for *refining* a goal by backward chaining. The `prim` and `thm` forms also declare tactics, all of which

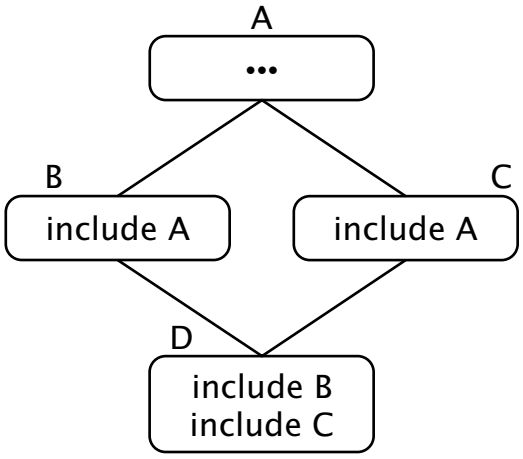

Figure 7.1: Multiple theory inheritance

are bound as ML values of the same name. All of the forms also take additional term arguments $p_1, \ldots, p_n$ that are used to specify extra arguments that may be needed for backward chaining. For instance, a declaration of the "cut" rule would require two extra arguments, one for new assertion, and another for its name.

$$\texttt{prim}\ \mathit{cut}\ x\ A : (a{:}\Gamma \vdash A)\ (b{:}\Gamma, x{:}A \vdash C) = (b[a/x] : \Gamma \vdash C)$$

The use of explicit parameters significantly simplifies rule application, which requires only first–order unification.

Formal modules are intended to be higher order, so there is no reason to create functors separately–a functor is just a function that produces an theory. However, functions on theories can be declared by using an `axiom` form in a theory implementation. The `axiom` is considered part of the signature of the theory, and it must be justified if the theory is instantiated using the `include` form.

7.7 Theory inheritance

The theories are object-oriented, in the sense that a theory specifies a *class* that can inherit rules and implementations from other classes. All rules and theorems that are valid in a superclass remain valid in the subclass. Syntactically, a theory declares itself to be a subclass of another class by using the "`include` *name*," where *name* is the name of a the-

ory signature. Operationally, an include directive treats the included theory is if it were inlined in the module, with one exception: modules are inlined at most once (an implicit sharing constraint). In the diagram above, we describe a scenario where modules *B* and *C* both include module *A*, and module *D* includes both *B* and *C*. Only *one* copy of *A* is inlined, and the modules *B* and share the common implementation.

The include directive is the logical equivalent of the open directive in OCaml. The include directive "opens" the logical namespace, and it establishes a logical dependence between modules. The general form form of the include directive is as follows:

```
include module-path
```

Multiple include directives may appear in interfaces and implementations, establishing multiple dependencies that establish a logical inheritance graph. For example, the following module uses the basic logical definitions, and reasoning about numbers to express a well-known problem in number theory:

```
include Itt_logic
include Itt_int

interactive fermat 'H :
  sequent ['ext] { 'H >-
    all i:  nat.  (i >= 3) =>
      not (exst a:  nat.  exst b:  nat.  exst c:  nat.
        power{a; i} + power{b; i} = power{c; i} in nat) }
```

The included modules Itt_logic and Itt_int provide the logical context. Itt_logic provides logical defiitions for all and exst, implication, and the Itt_int module provides reasoning about numbers. These modules have include directives of their own. The Itt_logic module contains the following logical dependencies:

```
include Itt_equal
include Itt_rfun
include Itt_dfun
include Itt_fun
include Itt_dprod
include Itt_prod
include Itt_union
include Itt_void
include Itt_unit
include Itt_struct
```

Each of the modules used in `Itt_logic` are recursively `include`'d in the Fermat theory. In general, an `include` recursively opens all ancestors of the module included. The `include` directive performs the following operations for each included module:

- All syntax declarations defined with `declare` are inherited.
- All display forms are inherited.
- All computational rewrites, axioms, and inference rules are inherited.

The `include` does **not** open the OCaml namespace. That operation must be performed separately for OCaml content, including tactics and rewrites.

7.7.1 Naming conflicts

Multiple modules may define operators with the same name. For example, two theories (say *A* and *B*) may both declare a conjunction

```
declare "and"{'a; 'b}
```

and they may impose different logical interpretations of the `"and"` term. The general rule for solving conflicts is the following:

- unqualified operator names refer to the most recent theory that defines the operator,
- operators names can be qualified to refer to other definitions.

A logic that relates the theories A and B must refer to terms declared in both *A* and *B*. *Qualified* names are expressed by module paths composed with the ! symbol. Here is an example, showing an interpretation of `A.and` as the "conditional" form of `B.and`. (The `"and"` operator is quoted because `and` it is an OCaml keyword.)

```
include A
include B
rewrite and_interp :  (A!"and"{'a; 'b}) <-->
    (B!"and"{'a; B!implies{'a; 'b}})
```

7.8 Rules, tactics, and resources

So far, we haven't spoken much about how theorems can be derived. The primitive formal framework contains a term language and allows the declaration of judgments. The primitive meta-logic contains a single rule, which is a restricted version of modus-ponens in a meta-sequent calculus. We will denote these meta-sequents with the turnstile $\models$ to distinguish them from sequents in the term language. The primitive inference rule allows a goal to be derived if the assumptions contain an assumed judgment that specifies the goal as its result, and all the antecedents of the judgment are derivable.

$$\frac{\Gamma, r\colon T_1 \longrightarrow \cdots \longrightarrow T_n, \Delta \models T_i \quad i \in \{1 \ldots n-1\}}{\Gamma, r\colon T_1 \longrightarrow \cdots \longrightarrow T_n, \Delta \models T_n}$$

Conceptually, during a proof, the assumption list contains all rules that have been previously declared in the theory containing the proof—including any rules declared in parent theories. Each `axiom` statement declares an infinite number of rules, one for each substitution instance. For instance, the statement `axiom` *hyp* $: \Gamma, x\colon A, \Delta \vdash A$ declares a rule for any variable for x, any term for A, and any term sequences for Γ and Δ.

As we stated in Section 7.6, when an axiom is declared, the framework automatically produces a tactic, which provides a handle that can be used request a primitive application of the rule. The tactic for and-introduction, when applied, would compute two subgoals, one for each branch of the conjunction. Tactics are associated with the theories in which they are declared, and the framework enforces the restriction by allowing the tactic to be applied only in proofs in sub-theories.

Tactics can also be combined with a few combinators (also called *tacticals*). The `andthen : tactic -> tactic -> tactic` tactical applies its first tactic, and then applies the second tactic to all the subgoals. The `orelse : tactic -> tactic -> tactic` applies the first tactic, and if it fails by raising a refinement exception, applies the second tactic. The `thenL : tactic -> tactic list -> tactic` tactical applies the first tactic, and then maps the tactic list across the subgoals, which must have the same number.

Because of the undecidability of proofs in higher-order logics, higher order theorem provers are typically designed to be interactive. The goals are presented to the user and interaction is typically by backward-chaining. However, in many cases the majority of inference steps in a formal proof are trivial, and a great deal of effort is exerted to automate the "obvious" steps in the proof with decision procedures or heuristics.

In a modular system, the task becomes more difficult because of

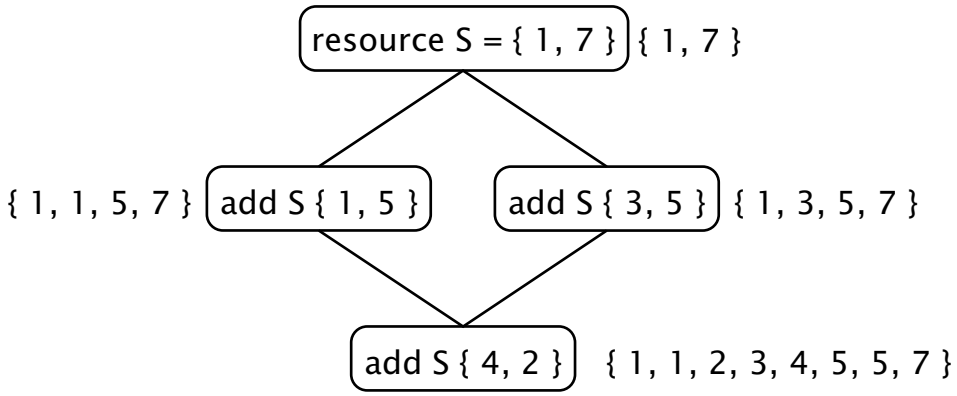

Figure 7.2: A “Set” resource

the potential for multiple distinct logics. In some modules proofs will be performed in a logic the does not include a a general collection of logical operators. As rules are added to a logic, decision procedures may change drastically (consider the case where the law of excluded middle is added intuitionistic propositional logic). We do not address the algorithmic changes here. Instead, we address the issue of modular proof automation with a device called *resources*.

Intuitively, a *resource* is a property of a type theory that can be produced by combining the resources of its parts. Syntactically, a resource is like a method that automatically includes the values of the methods it is inheriting. For example, the resource S in the diagram in the next paragraph computes the union of the multisets in each module. Note that the multiset in the root theory is inherited only once.

From the view of the implementation, a resource is really composed of three methods, `create`, `add`, and `join`. The `create` method creates an empty resource in a root theory. The `add` method adds a value to an inherited resource (in the diagram, the statement `add` S $\{1, 5\}$ computes the union of $\{1, 5\}$ and $\{1, 7\}$ and assigns it to the resource). The `join` method combines the value of a resource that is inherited from multiple ancestors (the final class computes the `join` of the sets $\{1, 1, 5, 7\}$ and $\{1, 3, 5, 6\}$ before the `add` S $\{4, 2\}$ statement).

Typically, resources are used to collect the parts of a modular reasoning method. For instance, a type theory may implement modules for each of its type constructors, and it might provide a decision procedure that has a component for each type constructor. These components would be collected in a resource, from which a tactic would ultimately be extracted. An example of such a resource is given in the first ex-

ample, where a decision procedure is generated for the intuitionistic propositional logic.

7.9 The MetaPRL compiler and logical library

At this point, we have discussed the content of logical modules. But how are these modules represented in the implementation? Referring back to Figure 6.2, the *logical library* provides long-term storage of modules as well as proofs that accompany the declarations of theorems in a module.

The logical library acts as the central storage repository, and it is accessed by the MetaPRL compiler, the editor, and the theorem prover. In its simplest form, the logical library provides a kind of filesystem containing three kinds of objects. Each object has a name, and content with the following descriptions.

1. Module *definitions*, described in this Chapter, are represented as objects with names ending with the suffix `.mli` (for signatures) or `.ml` (for structures). A module definition is treated as a sequence of characters without interpretation. A module in this form may be compiled by the MetaPRL compiler `prlc` to parse the character string and extract and summarize its logical content.

2. Module *summaries* are represented as objects with names ending with the suffix `.cmiz` (for signatures) or `.cmoz` (for structures). Module summaries are the *working versions* of modules used by the compiler, editor, and theorem provers as a means of interchange of logical content. The main difference between a summary and the original source content is that the summary has been *parsed* so that its logical content is accesible to the MetaPRL suite of tools. The MetaPRL compiler is used to produce a summary for a source file. The editor can be used afterwards to browse, modify, and add proofs to the summary. The theorem prover uses the syntax and rules in a summary to justify the proofs, and it uses tactics defined in the summary to provide proof automation.

3. Program *executables* are also stored in the library. Once all the modules in a program have been compiled to summaries, and all necessary proofs have been provided using the editor, the MetaPRL compiler can be used to link the modules into a program executable.

The logical library has two implementations. The simplest implementation uses the standard filesystem. Standard text editor can be

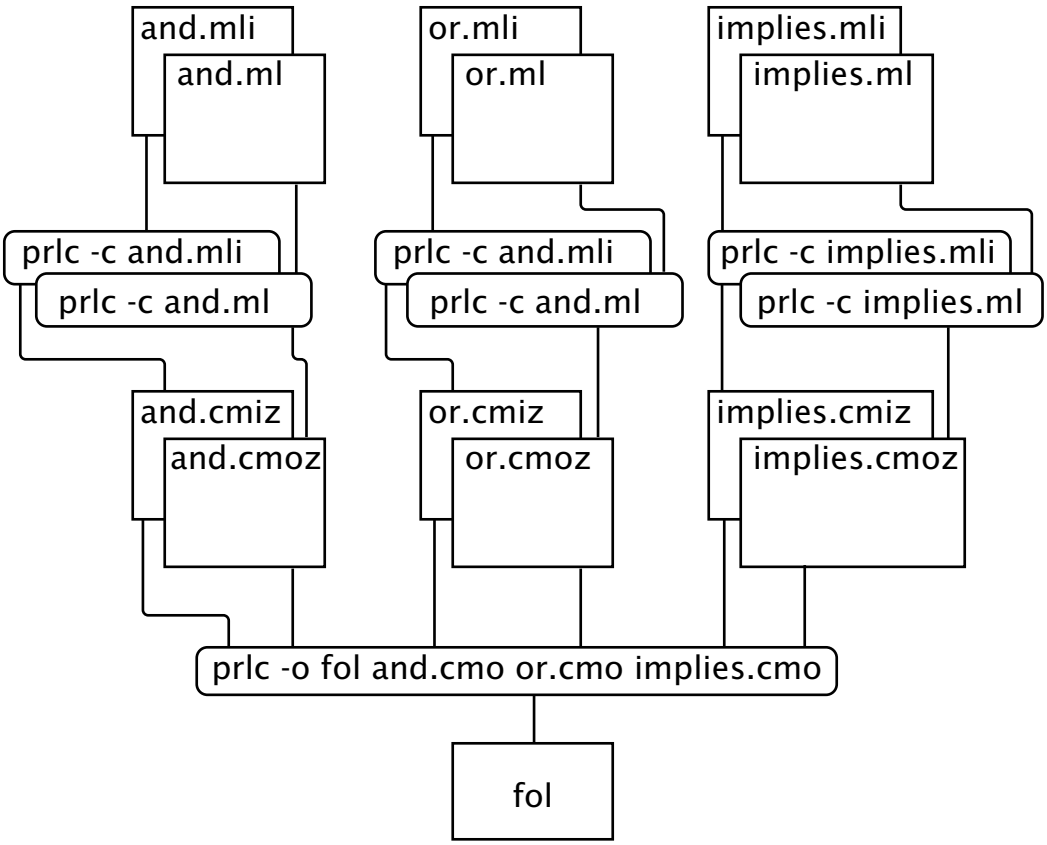

Figure 7.3: An example of compiling first-order logic

used to produce source `.mli` and `.ml` files. Once they have been created, the MetaPRL compiler `prlc` can be used to compile them to logical summaries as `.cmiz` and `.cmoz` files. The MetaPRL compiler is an extension of the OCaml compiler, and the usage is identical: to compile a `.mli` file (say `fol.mli`, which describes first-order logic), the usage is `prlc -c fol.mli`. The "`-c`" option specifies that the compiler is to produce a summary, without linking. The same usage hold for an implementation file `fol.ml`. Once these steps have been performed successfully, the filesystem will contain two new files `fol.cmiz` that holds the summary of the signature, and `fol.cmoz` that holds the summary of the implementation. All *parent* modules, defined by any `open` or `include` statements in the module, must already be compiled.

Once the summary is created, the MetaPRL editor may be used to browse and edit the module, updating the summary files in place. In the final step, the `prlc` compiler may be used again to link the `.cmoz` files into an executable.

The logical library also serves as a central storage point for communication with other theorems provers. At present, this connection is under development, but the objective is to provide a communication link between multiple provers by defining translations between the syntax of the logcal summaries of each. This communication mechanism uses the MathBus [Zip] and the Nuprl5 [ACE+00] library to establish the connection between MetaPRL and MathBus.

Chapter 8

The MetaPRL Refiner

The MetaPRL system uses *tactics* to automate proof search. Tactics are informal procedures written in OCaml that perform both algorithmic and heuristic proof search, as well as computationally intensive domain-specific proof procedures. In general, tactic-based provers perform worse than fully-automated special-purpose provers. The MetaPRL refiner uses a new modular proving architecture that significantly increases the speed of the core logic engine. The speedup is due to efficient data structures and *modularity*, which allows parts of the prover to be customized on a domain-specific basis. The MetaPRL prover obtains speedups of more than two orders of magnitude over traditional Nuprl prover.

8.1 Introduction

Several provers [CNSvS94, COR+95, DFH+93, CAB+86, GM93, Hic97, Pau94] use higher-order logics for reasoning because the expressivity of the logics can be used to obtain concise models, and because they can express meta-principles that allow results to be re-used on multiple problems. Proof automation is coded informally in a *meta-language* (often a variant of ML) as *tactics*. Automation speed has a direct impact on the level of reasoning. If proof search is slow, more interactive guidance is needed to prune the search space, leading to excessive detail in the tactic proofs.

The MetaPRL prover achieves more than two orders of magnitude speed-up over the existing Nuprl4 implementation. We obtain the speedup in two parts: our architecture is modular, allowing components to be replaced with domain-specific implementations, and we use efficient data structures to implement the proving modules.

In this chapter, we explain this modular architecture. We show that

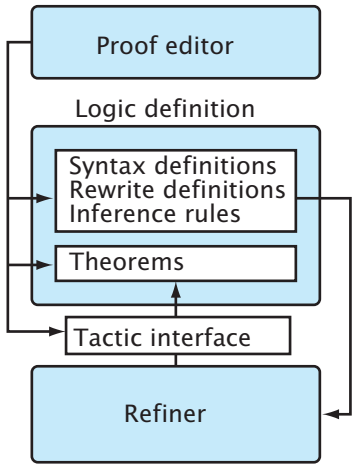

Figure 8.1: General tactic prover architecture

the logic engine can be broken into three modules: a *term* module that implements the logical *language*, a term *rewriter* that applies primitive inferences, and a *proof* module that manages proofs and defines tactics. The computational behavior of proof search is dominated by term rewriting and operations on terms, and we present implementations of the modules for domains with frequent applications of substitution (like type theory), and for domains with frequent applications of unification (like first-order logic).

The organization of the chapter is a follows. In Section 8.2, we give an overview of tactic proving, and present the high-level architecture. In Sections 8.3, 8.4, and 8.5, we explore the proving modules in more detail, and develop their implementations. In Section 8.6, and we compare the performance of the different implementations. This work builds on the efforts of many systems over the last decade, and in Section 8.7 we present related work on proof automation.

8.2 Architectural Overview

The general MetaPRL architecture can be divided into three parts, shown in Figure 8.1. A *logic* contains the following kinds of objects:

1. *Syntax definitions* define the *language* of a logic,
2. *Inference rules* define the primitive inference of a logic. For instance, the first-order logic contains rules like AXIOM and MODUS_PONENS in a sequent calculus.

$$\frac{}{\Gamma, a{:}A \vdash A}\ \text{AXIOM} \tag{8.1}$$

$$\frac{\Gamma, f{:}A \Rightarrow B \vdash A \quad \Gamma, f{:}A \Rightarrow B, b{:}B \vdash C}{\Gamma, f{:}A \Rightarrow B \vdash C}\ \text{MODUS_PONENS} \tag{8.2}$$

3. *Rewrites* define computational equivalences. For example, the Nuprl type theory defines λ abstraction and application, with the following equivalence:

$$(\lambda x.b)\ a \longleftrightarrow b[a/x]. \tag{8.3}$$

4. *Theorems* provide proofs for derived inference rules and axioms.

The *refiner* [Bat79] performs two basic operations. First, from the parts of the logic, it builds the reasoning primitives:

1. Syntax definitions are compiled to sentence constructors and destructors.

2. Rewrite primitives (and derived rewrite theorems) are compiled to *conversions* that allow computational reductions to be applied to terms in a proof.

3. Inference rules and theorems are compiled to primitive *tactics* for applying the rule, or instantiating the theorem.

The second function of the refiner is to implement the *application* of conversions and tactics, producing justifications from the proofs. The major parts of the refiner interface are shown below. It defines abstract types for data structures that implement terms, tactic and rewrite definitions, proofs, and logics. Proof search is performed in a backward-chaining goal-directed style. The `refine` function takes a `logic` and a `tactic` search procedure, and applies it to a *goal* term to produce a *partial* `proof`. The goal and the resulting *subgoals* can be recovered with the `sub/goal_of_proof` projection functions. Proofs can be composed with the `compose proof subproofs` function, which requires that the goals of the `subproofs` correspond to the subgoals of the `proof`, and that both derivations occurred in the same logic. If an error occurs in any of the refiner functions, the `RefineError` exception is raised. The `tactic_of_conv` function creates a tactic from a rewrite definition. The final two functions, called *tacticals*, are the primitives for implementing proof search. Operationally, the `andthen tac1 tac2` tactic applies `tac1` to a goal and immediately applies `tac2` to all the subgoals, composing the result. The `orelse tac1 tac2` is equal to `tac1` on goals where `tac1` does not produce an error, otherwise it is equivalent to `tac2`.

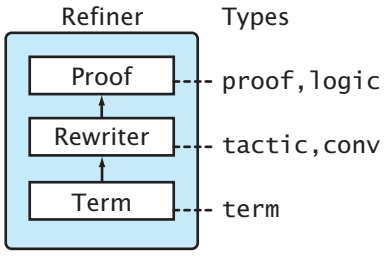

Figure 8.2: Layered refiner

```
module type RefinerSig = sig
  type term, tactic, conv, proof, logic
  exception RefineError
  val refine :  logic → tactic → term → proof
  val goal_of_proof :  proof → term
  val subgoals_of_proof :  proof → term list
  val compose :  proof → proof list → proof
  val tactic_of_conv :  conv → tactic
  val andthen :  tactic → tactic → tactic
  val orelse :  tactic → tactic → tactic
end
```

The *proof editor* provides the user interface. It allows logical components to be viewed and modified, it manages the construction of tactic proofs, and it organizes the presentation of the logic primitives and proofs.

In a prover like Nuprl4, the refiner is *monolithic*. There is no well-defined separation of the refiner into components, and there is no well-defined interface like the `RefinerSig` we defined above—there is *one* built-in refiner. The first architectural design choice we make is simple: if we partition the refiner into abstract parts, we can create domain-specific implementations of its parts. The choice of partitioning we use is guided by the *type* definitions, producing the layered architecture shown in Figure 8.2. The lowest layer, the *term* module, defines the logical language; the *rewriter* module implements applications of primitive tactics and conversions using term rewriting; and the *proof* module defines the `logic` and `proof` data types. We present specifications and implementations of these modules in the following section.

8.3 The *term* module

All logical terms, including goals and subgoals, are expressed in the language of *terms*, implemented by the *term* module. The general syntax of terms was described in Chapter 5. Each term has 1) an operator-name (like "sum"), which is a unique name indicating the logic and component of a term; 2) a list of parameters representing constant values; and 3) a set of possibly-bound subterms. The following syntax is used to describe terms.

$$\underbrace{opname}_{operator\ name}\ \underbrace{[p_1;\cdots;p_n]}_{parameters}\underbrace{\{\vec{v_1}.t_1;\cdots;\vec{v_m}.t_m\}}_{subterms}$$

The term module implements several basic term operations: substitution ($b[a/x]$) of a term (a) for a variable (x) in a term (b), free-variable calculations, α-equivalence, etc. When a logic defines a rule, the refiner compiles the rule pattern into a sequence of term operations. The term interface is shown in Table 8.1. The abstract types `opname`, `param`, `term`, and `bound_term` represent operator names, constant parameters, terms, and bound terms (the subterms of a term). The major operations include destructors to decompose terms and bound-terms, as well as a substitution function `subst`, free variable calculations, and term equivalence.

8.3.1 Naive term implementation (`Term_std`)

The most immediate implementation of terms is the naive "standard" implementation, which builds the term with tupling.

```
type opname = string list
and param = Int of int | String of string
and term = opname * param list * bound_term list
and bound_term = string list * term
```

While this structure is easy to implement, it suffers from poor substitution performance. The pseudo-code in Table 8.2 gives an outline of the substitution algorithm.
The `sub` argument is a list of `string/term` pairs that are to be simultaneously substituted into the term in the second argument. The main part of the substitution algorithm is in the part for substituting into bound terms. In step ①, the substitution is modified by removing any `string/term` pairs that are freshly bound by the binding list `vars`, and in step ②, the binding variables are renamed if they intersect with any of the free variables in the terms being substituted.

Table 8.1: Term module signature

```
module type TermSig = sig
  (* Types and constructors: *)
  type opname, param, term, bound_term
  val mk_opname : string list ->opname
  val mk_int_param : int ->param
  val mk_string_param : string ->param
  val mk_term : opname ->
    param list ->bound_term list ->term
  val mk_bterm : string list ->term ->bound_term

  (* Destructors and other operations: *)
  val dest_term : term ->
    opname * param list * bound_term list
  val dest_bterm : bound_term ->string list * term
  val subst : (string * term) list ->term ->term
  val free_vars : term ->string list
  val alpha_equal : term ->term ->bool
end
```

Table 8.2: Substitution for Term_std

```
let rec subst sub t =
  if t is a variable then
    if (t, t') ∈ sub then t' else t
  else let (opname, params, bterms) = t in
    (opname, params, List.map (subst_bterm sub) bterms)
and subst_bterm sub (vars, t) =
  let sub' = remove (v, t') from sub if v ∈ vars in  ①
  let vars', sub'' =
    rename binding variables to avoid capture in      ②
  (vars', subst sub'' t)
```

Roughly analyzed, this algorithm takes time at least linear in the size of the term on which the substitution is performed. Furthermore, each substitution performs a full copy of the term. Substitution is a very common operation—each application of an inference rule involves at least one substitution. The next implementation performs lazy substitution, useful in domains like type theory.

8.3.2 Delayed substitution (`Term_ds`)

If substitution is frequent, it is often more efficient to save computations for use in multiple substitution operations. We use three main optimizations: we save free-variable calculations, we perform lazy substitution, and we provide special representations for commonly occurring terms like variables and sequents.

When a substitution is performed in a term for the first time, we compute the set of free variables of that term, and save them for later use. When a substitution is applied, the free-variables set is used to discard the parts of the substitution for variables that do not occur free in the term. This saves time, and it also saves space by preserving subterms where the substitution has no effect.

During proof search, tactic applications usually fail, and usually only part of the substitution result is examined in the proof search. In this common case, it is more efficient to delay the application of a substitution until the substitution results are actually requested by the `dest_term` function.

We also optimize two commonly-occurring terms: *variables* and *sequents*. Rather than using the term encoding of variables, we provide a custom representation using a string. The *sequent* optimization uses a custom data structure to give constant-time access to the hypotheses, instead of the usual linear-time encoding. These "custom" terms are abstract optimizations—they do not change the `Term` interface definition. For each custom term, we add special-case handlers to each of the generic term functions.

The definition of terms in Table 8.3 uses all of these optimizations (the definitions for the `bound_term`, `opname` and `param` types are unchanged). The definition of sequents uses arrays to represent the hypotheses and conclusions of the sequent.

The `free_vars` field caches the free variables of the term, using `VarsDelayed` as a placeholder until the variable set is computed. The `core` field stores the term value, using the `Term` variant to represent values where a substitution has been expanded, the `Subst` variant to represent delayed substitutions, and the `Var` and `Sequent` variants for custom terms.

Table 8.3: Terms with delayed substitution

```
type term = { free_vars :    VarsDelayed
                           | Vars of string set;
  core:  Term of (opname * param list * bound_term list)
         | Subst of (subst * term)
         | Var of string
         | Sequent of sequent }
and subst = (string * term) list
and sequent = (string * term) array * term array
```

The free-variables computation, shown in Table 8.4, is one of the more complex operations on this data structure. When the free variables are computed for a term, there are three main cases: if the free variables have already been computed, they are returned; if the `core` is a `Term`, the free variables are computed from the subterms; and if the `core` is a delayed substitution, the substitution is used to modify the free variables of the inner term.

If the free variables haven't already been computed, the `free_vars` function computes them, and assigns the value to the `free_vars` field of the term. In the `Term` case, the free variables are the union of the free variables of the subterms, where any new binding occurrences have been removed. In the `Subst (sub, t)` case, the free variables are computed for the inner term `t`, then the variables being replaced are removed from the resulting set, and then the free variables of the substituted terms are added.

The `subst` function, shown in Table 8.5, has a simple implementation: eliminate parts of the substitution that have no effect, and save the result in a `Subst` pair if the resulting substitution is not empty.

The `set` implementation determines the complexity of substitution. If the `set` lookup takes $O(1)$, then pruning ① takes time linear in the number of variables in `sub`.

The effect of the substitution is delayed until the point that the term is destructed. The `dest_term` function is required to expand the substitution by one step. We use the `get_core` function, shown in Table 8.6, to expand the toplevel substitutions in the term.

When the substitution is applied to a `Term`, it is recursively applied to the subterms, performing a single substitution step. Note that the `List.assoc` in the `Var` case will never fail, due to two invariants: substitution lists are never empty, and the domain of the substitution is contained within the free-variables of the term.

Table 8.4: Free variable calculation

```
let rec free_vars = function
    { free_vars = Vars fv } ->fv
  | { core = core } as t ->
    let fv = match core with
        Term (_, _, bterms) ->
         Set.map_list free_vars_bterm bterms
      | Subst (sub, t) ->
         free_vars_subst sub (free_vars t)
      | Var v ->Set.singleton v
      | Sequent (hyps, concls) ->
         free_vars_sequent 0 hyps concls
    in (t.free_vars ← Vars fv); fv
and free_vars_bterm (bvars, t) =
  Set.subtract_list (free_vars t) bvars
and free_vars_sequent i (hyps, concls) =
  if i = Array.length hyps then
    Set.map_array free_vars concls
  else
    let (v, h) = hyps.(i) in
      Set.subtract v
        (free_vars_sequent (succ i) hyps concls)
and free_vars_subst sub fv =
  Set.union
    (Set.subtract_list fv (List.map fst sub))
    (Set.map_list free_vars (List.map snd sub))
```

Table 8.5: Delayed substitution algorithm

```
let subst sub t =
  let fv = free_vars t in
    match remove (v, t') from sub if v ∉ fv with ①
      [] ->t (* substitution has no effect *)
    | sub' ->{ free_vars = VarsDelayed;
      core = Subst (sub', t) }
```

Table 8.6: Substitution expansion for Term_ds

```
let rec get_core = function
  { core = Subst (sub, t') } as t ->
  let core' = match get_core t' with
      Var v ->get_core (List.assoc v sub)
    | Term (opname, params, bterms) ->
      Term (opname, params,
        List.map (do_bterm_subst sub) bterms)
    | Sequent seq ->Sequent (sequent_subst sub seq)
  in (t.core ← core'); core'
| { core = core } ->core
and do_bterm_subst sub (vars, t) =
  let sub' = remove (v, t) from sub if v ∈ vars in
  let vars', sub'' =
    rename binding variables to avoid capture in
    (vars', subst sub'' t)
```

Table 8.7: Term expansion for Term_ds

```
let rec dest_term t =
  match get_core t with
      Term (opname, params, bterms) ->
      (opname, params, bterms)
    | Var v ->(mk_opname ["variable"], [String v], [])
    | Sequent s ->dest_sequent s
```

The `dest_term` function, shown in Table 8.7, first uses `get_core` to expand all substitutions, and then it returns the parts of the term. To preserve the external interface of the term module, it is also required to convert the custom terms back to their original form (as usual, we omit the code for sequents).

8.4 The *rewriter* module

All rule applications and term rewrites in MetaPRL are implemented using term rewriting module. In Nuprl4 [Jac94], rule application is implemented using a rule *interpreter*, and computational rules for term rewriting are built into the refiner. In MetaPRL, efficient term rewriting

Table 8.8: Signature for the rewrite module

```
module Rewrite (Term : TermSig) : sig
  type redex_prog, con_prog, state
  exception RewriteError
  val compile_redex : term ->redex_prog
  val apply_redex : redex_prog ->term ->state
  val compile_contractum : redex_prog ->term ->con_prog
  val build_contractum : con_prog ->state ->term
end
```

is key to achieving high levels of performance, and the rewrite module uses a *rewrite compiler* to compile rules and rewrites to a *virtual rewrite machine*. The machine is based on the same ideas as the Categorical Abstract Machine (CAM) [CCM85], but the machine language is tailored for optimized term rewriting.

As explained in Section 5.2, inference rules and computational rewrites are expressed using second-order patterns. For example, the rewrite for beta-reduction is expressed with the following pattern:

$$(\lambda x.b_x)\ a \rightarrow b_a$$

In this rewrite, the variable a is a *pattern* variable, representing the "argument" term. The variable b_x is a *second-order pattern* variable, representing a term with a free variable x. The pattern b_a represents a *substitution*, with a substituted for x in b. The $(\lambda x.b_x)\ a$ is called the *redex*, and the substitution b_a is called the *contractum*.

The rewriter module provides four major functions, shown in Table 8.8. The `compile_redex` function takes a redex pattern, expressed as a term, and it compiles it to a redex *program*. The `apply_redex` function applies a pre-compiled program to a specific term, raising the `RewriteError` exception if the pattern match fails, or returning a `state` that summarizes the result. The `compile_contractum` compiles a contractum pattern against a particular redex program, and the `build_contractum` term takes the contractum program and the result of a redex application, and produces the final contractum term.

Currently, we use a single implementation of the rewrite module, implementing the `Rewrite` interface using a *virtual rewrite machine*. Redices are compiled to bytecode programs that perform pattern matching, storing the parts of the term being matched in several register files. Contracta are also compiled to bytecode programs that construct the contractum term using the contents of the register file.

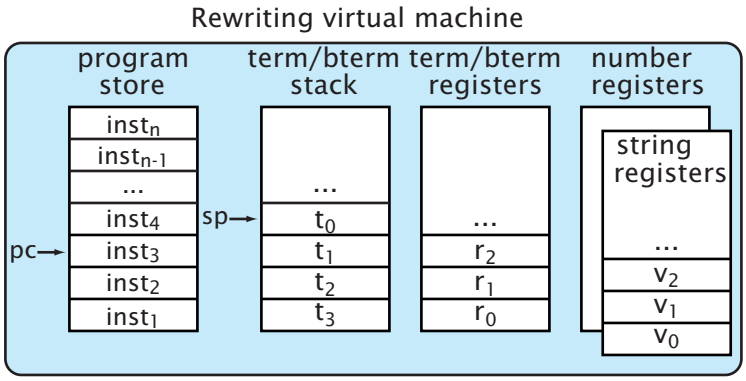

Figure 8.3: Rewrite virtual machine

The virtual machine has the four parts shown in Figure 8.3:

1. a *program* store and program counter for the rewrite program,
2. a *term/bterm stack* with a stack pointer to manage the current term being rewritten,
3. a *term/bterm* register file,
4. a *parameter* register file for each type of parameter.

The instructions for the machine are shown in Figure 8.9. The matching instruction `dest_term` checks if the term at the top of the term stack has the operator name `opname`, and if it has the right number of bound terms and parameters of the given types. If it succeeds, the parameters are saved in the parameter registers, the bound terms are pushed onto the bterm stack, and the term is popped from the term stack. The `mk_term` instruction does the opposite: it retrieves the parameter values from the register file, pops `bterm_count` bound terms from the bterm stack, adds the `opname` and pushes the resulting term on the term stack. The `dest_bterm` and `mk_bterm` function perform similar operations on bound terms.

The `so_var` instruction copies the term on the term stack to a term register, along with the free variables $v_1, \ldots, v_n$. The corresponding constructor `so_subst` pushes the term in register r, with literal terms $t_1, \ldots, t_n$ substituted for its substitution variables $v_1, \ldots, v_n$ (saved with the term in the term register). The `match_term` instruction is used during matching for redices like $(x + x) \rightarrow 2x$ that contain common subterms.

Table 8.9: Virtual machine instructions

Matching:

dest_term	opname$[p_1;\cdots;p_n]$*bterm_count*
dest_bterm	$v_1,\ldots,v_n.r$
match_term	$r[t_1;\cdots;t_n]$
so_var	$r[v_0;\cdots;v_n]$

Constructors:

mk_term	opname$[p_0;\cdots;p_n]$*bterm_count*
mk_bterm	$v_1,\ldots,v_n.r$
so_subst	$r[t_1;\ldots;t_n]$

p_i: parameter register (*string* or *number*)
v_1: string register
r: term register
t_i: literal term

8.5 The *proof* module

The third part of the refiner manages validity in logics as well as maintaining proof trees for theorems. The proof module exports the interface shown in Table 8.10. The `empty_logic` is the logic without any rules/rewrites. The `join_logics` function builds the union of two logics, and the `add_rule` and `add_rewrite` function add rules/rewrites from their syntactical description as terms. The `proof` type represents a partial proof tree [CKB84], which may be modified by applying a tactic to the proof goal with the `refine` function. The `compose` function is used to stitch together partial proofs into larger proofs. Accounting must be performed here—the proofs being joined must belong to the same logic. If an error occurs in any of the functions, the `RefineError` exception is raised. These functions are not difficult to implement, and we skip the description of their implementations.

8.6 Performance

We group the performance measurements into two parts. All measurements were done on a Linux 400MHz Pentium machine, with 512MB of main memory, and all times are in seconds. For the first part, we compare the speed of the MetaPRL prover (using the modular refiner) with

Table 8.10: The proof module signature

```
module Proof (Term :  TermSig) (Rewrite :  RewriteSig) :
sig
  type logic, tactic, rewrite, proof, validation
  exception RefineError
  val empty_logic :  logic
  val join_logics :  logic ->logic ->logic
  val add_rule :  logic ->term ->logic * tactic
  val add_rewrite :  logic ->term ->logic * rewrite
  val new_proof :  term ->proof
  val refine :  proof ->tactic ->proof
  val compose :  proof ->proof list ->proof
  val proof_goal :  proof ->term
  val proof_subgoals :  proof ->term list
  val check :  proof -> validation
end
```

the Nuprl4 prover. For the first example, we perform pure evaluation based on the following definition of the factorial function:

$$\text{rewrite } \mathit{fact}\{i\} \longleftrightarrow \mathit{if}\ i = 0\ \mathit{then}\ 1\ \mathit{else}\ i * \mathit{fact}\{i-1\}$$

We used the following evaluation algorithm: recursively traverse the term top-down, performing beta-reduction, unfolding the `fact` definition (taking care to evaluate the argument first), etc. This algorithm stresses search during rewriting. Roughly speaking, evaluation should be quadratic in the factorial argument: each term traversal is linear in the size of the term, and the size of the term grows linearly with each traversal (rewriting does not use tail-recursion), until the final base case is reached and the value is computed. The following table lists the performance numbers.

	Argument value			
Configuration	100	250	400	650
`Term_std`	0.35	2.25	6.0	17.5
`Term_ds`	0.4	2.7	7.5	21
Nuprl4	55	330	*	*

On this example, Nuprl4 took between 120 and 150 times longer on the problems where it finished within 30 minutes. On the two

larger problems, we terminated the computation after 30 minutes.[1] In MetaPRL, the largest problem performs about 20 million attempted rewrites.

This table also shows an interesting difference between the term module implementations. The "naive" term module performs better on this example because the recursive traversals of the term expand most of the delayed substitutions.

There are several reasons why Nuprl4 is slower on these problems. First, Nuprl4 tactics are written in the LCF meta-language "classic ML" [GMW79], which is compiled to Common Lisp; the Nuprl4 refiner is also implemented in Common Lisp. In contrast, the entire MetaPRL system is implemented in OCaml. Programs in OCaml can be compiled directly to machine code, and our experiments show a typical performance improvement over classic ML of about a factor of five. The second reason for MetaPRL's performance gain is the use of highly optimized data structures, including an efficient term module implementation that includes lazy substitution; whereas Nuprl4 is relatively unoptimized. Finally, we have devoted a lot of time to improving MetaPRL's overall performance and optimizing key algorithms and data structures. Nuprl4 has not been put through a critical performance analysis. This is due in part to Nuprl4's monolithic implementation, which makes it difficult to perform a careful design analysis. In our experience, the modular MetaPRL implementation has made it easier to isolate and optimize parts of the system.

The next example also compares MetaPRL with Nuprl4, on a the pigeonhole problem stated in propositional logic. The pigeonhole problem of size i proves that $i + 1$ "pigeons" do not fit into i "holes." The `pigeonT` tactic performs an efficient search customized to this domain, and the `propDecideT` tactic is a generic decision procedure for *intuitionistic* propositional logic (based on Dyckoff's algorithm [Dyc92]). Both search algorithms explore an exponential number of cases in i.

		Problem size			Memory
Configuration	Tactic	2	3	4	(Max MB)
`Term_std`	`pigeonT`	<0.1	2.46	93.8	170
`Term_ds`	`pigeonT`	<0.1	0.59	15.1	32
`Term_std`	`propDecideT`	0.28	233	>1800	
`Term_ds`	`propDecideT`	0.11	54	>1800	
Nuprl4	`pigeonT`	0.5	89	>1800	
Nuprl4	`propDecideT`	21.9	>1800	>1800	

[1] Nuprl4 *can* evaluate these terms. The built-in term evaluator, which is independent of the refiner, evaluates the largest example in about 22 seconds.

In this example, the delayed-substitution implementation of terms performs significantly better than the naive implementation, partly because of the efficient substitution in the application of the rules for propositional logic, and also because the `Term_ds` module preserves a great deal of sharing of common subterms. On the largest problem the `pigeonT` tactic performs about 1.5 million primitive inference steps.

For the last examples, we give a few comparisons between the MetaPRL modules in three domains. The GEN problems is a heredity problem in a large first-order database. The NUPRL and CZF problems are problems synthesized from the Nuprl type theory, and the CZF set theory. These last two examples are replays of proof transcripts in the higher-order theories. We have combined transcripts from several proofs, and used repetition to expand the proofs to 10 times their original size (the original proof transcripts replayed too quickly to get significant measurements). The transcripts contain a mix of low-level proof steps, such as lemma application and application of inductive reasoning, to higher-level steps that include verification-condition automation and proof search. The expanded transcripts correspond to 2,700 interactive steps in the NUPRL example, and 2,300 in CZF.

We don't include performance measurements for Nuprl4 on these examples, because the system differences require a porting effort (for instance, Nuprl4 does not currently implement a generic first-order proof search procedure). In our experience with Nuprl4, proofs with several hundred nodes tend to take several minutes to replay.

	Problem		
Configuration	GEN	NUPRL	CZF
`Term_std`	9.71	26.5	8.08
`Term_ds`	6.75	10.17	5.0

Once again, the `Term_ds` module performs better than the naive terms, due to the frequent use of substitution on the application of the rules of these theories. The first-order problem, GEN, performs proof search by resolution, using the refiner to construct a primitive proof tree only when a proof is found. This final step is expensive, because each resolution step has to be justified by the refiner. The final successful proof in this problem performs about 41 thousand primitive inference steps.

8.7 Related work

Harrison's HOL-Light [Har96] shares some features with the MetaPRL implementation. Harrison's system is implemented in Caml-Light, and

both systems require fewer computational resources than their predecessors. Howe [How98] has taken another approach to enhancing speed in Nuprl4. The programming language defined by the Nuprl type theory is *untyped*, leading to frequent production of well-formedness (verification) conditions. Using type annotations, Howe was able to speed up rewriting in Nuprl4 by a factor of 10.

Basin and Kaufmann [BK91] compare the Nuprl3 system and NQTHM [BM79] (the predecessor of the ACL2 [KM97] system). The NQTHM prover uses a quantifier-free variant of Peano arithmetic and list theory. Basin and Kaufmann's measurements showed that NQTHM was roughly 15 times faster for *different formalizations* of Ramsey's theorem.

Chapter 9

The MetaPRL Distributed Prover

Higher-order logics are expressive tools. Because of their complexity, proofs in higher-order logics often use a combination of interactive proving together with computationally-intensive tactic applications that perform proof automation. As problems and proof automation become more sophisticated, these proofs represent substantial investments—each interactive step may represent several hours of design time.

MetaPRL implements a distributed prover to address the problems of speed, availability, and reliability. The implementation supports arbitrary process joins and all-but-one process failures at any time during a proof. Proof distribution is completely transparent; the existing tactic base is unmodified.

9.1 Introduction

Proofs in higher-order logic can be computationally complex. It is easy for proof search to wander into areas of intractability, and many systems like HOL [GM93], Coq [DFH+93], Isabelle [Pau94], and Nuprl [CAB+86] combine interactive guidance with automated proving. While this model has been successful, it is expensive. Many formalizations require years of processing time and human effort. The expense, especially the human part, is the major obstacle preventing widespread adoption of these practices.

How can we ease the burden of proof? The MetaPRL prover aims at increasing the power of proof automation by exploiting widely available computational resources. The MetaPRL refiner implements distributing tactic proving over large groups of processors using the Ensemble group communication system. To counteract problems of reliability in distributed environments, we use Ensemble's fault recovery support to allow arbitrary reconfigurations at any time in a proof (if at least one

$a{:}A, f{:}A \Rightarrow B \vdash B$
```
refine fol (modus_ponens "f")
```

$a{:}A, f{:}A \Rightarrow B \vdash A$
```
refine fol (axiom "a")
```

$a{:}A, f{:}A \Rightarrow B, b{:}B \vdash B$
```
refine fol (axiom "b")
```

Figure 9.1: Example propositional proof

process remains running at any given time).

To preserve the large existing tactic base, we replace the existing tactic implementation with a functionally equivalent distributed tactic scheduler. No extra knowledge is required to use distributed proving, and our measurements show significant speedup for medium-sized groups.

In the remainder of the chapter, we introduce the distributed architecture in three steps: in Section 9.2 we review the properties of tactic proving needed for understanding the distributed prover; in Section 9.3 we generalize the tactic model to multi-threaded proving; and in Section 9.4 we present the communication model and the distributed proving architecture. In Section 9.5 we present some performance results for some of the logical domains in MetaPRL, including first-order logic and the Nuprl type theory. We summarize our results in Section 9.6 and we finish with a discussion of related work in Section 9.7.

9.2 Refinement Architecture and Tactics

Like other foundational LCF-style tactic provers such as Coq [DFH+93] and Nuprl [CAB+86], MetaPRL performs proofs in a backward-chaining goal-directed style. Given a *goal* sentence like the sequent $A, A \Rightarrow B \vdash B$, the user selects a *tactic* to apply to the sentence in an attempt to either prove it, or reduce it to a set of simpler *subgoals*. When a tactic is applied, it reduces the proof to the set of primitive inferences defined by the logic. For example, a proof of the sequent above is given in Figure 9.1. The `dT` tactic is a generic tactic for applying introduction and elimination rules, and the `trivialT` tactic applies "trivial" rules like the AXIOM rule of propositional logic.

As mention in the previous chapter, the proving architecture is im-

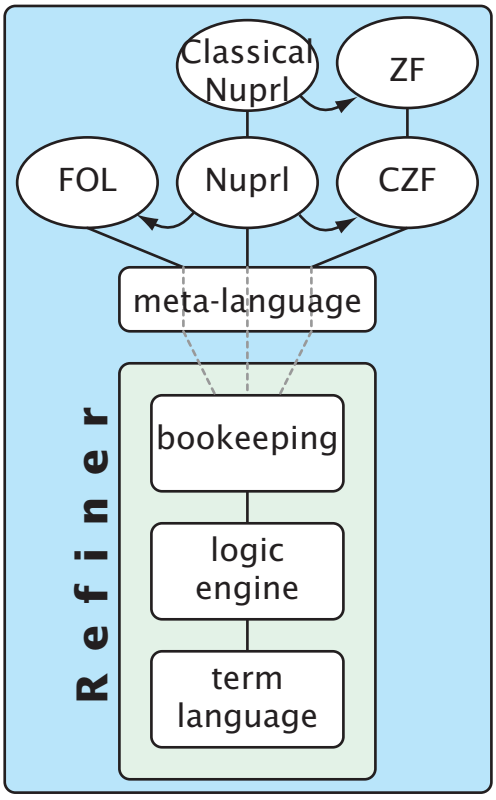

Figure 9.2: Tactic-prover architecture

plemented in several parts, shown in Figure 9.2. For each logic, there is a module defining the primitive rules of the logic. The *refiner*, discussed in Cahpter 8, defines tactics for applying a primitive rule to a goal to reduce it to a (possibly empty) set of subgoals. On top of this, there is a *primitive tactic* module that defines the basic tactic operations. Proof automation is implemented in the *tactic library*.

The tactic-primitives module simplifies the application of the refiner's rules by supporting proof search and composition with *tacticals*. The tactic interface is defined as follows:

```
module TacticPrimitives (Refiner) :  sig
  type prim_tactic, proof
  type tactic = term → prim_tactic
  val tactic_of_rule :  Refiner.rule → tactic
  val proof_of_goal :  logic → term → proof
  val refine :  proof → tactic → proof
  val andthenT : tactic → tactic → tactic
  val orelseT : tactic → tactic → tactic
end
```

The tactic-primitives module re-implements the `proof` type to con-

Table 9.1: Proof procedure for propositional logic

```
(* Refiner declarations:
* val imp_elim :  int → tactic
* val imp_intro :  tactic
* val axiom :  int → tactic
*)
let rec autoT t =
  orelseT (onSomeHypT 0 imp_elim)
    (orelseT (onSomeHypT 0 axiom) imp_intro)
and onSomeHypT tac i t =
  if i = hyp_count t then
    raise RefineError
  else
    orelseT (tac i thenT autoT)
      (onSomeHypT tac (i + 1)) t
```

tain single node proof trees. It contains a constructor for building tactics from rules, and it implements a `refine` function that works with tactics. The `tactic` type itself is implemented as a function that returns a primitive reasoning operation, given the goal the tactic is being applied to. The tactic module also implements two *tacticals:* the `(andthenT tac1 tac2)` tactic applies `tac1` to the goal, then applies `tac2` to each of the subgoals. The `(orelseT tac1 tac2)` tactic applies `tac1` to the goal. If it succeeds, the proof is returned; otherwise, `tac2` is applied instead.

When embedded in a meta-language like OCaml, these primitives are sufficient to encode proof search. For example, a tactic to prove propositional sentences with implication is shown in Figure 9.1. The `axiom`. `imp_elim`, and `imp_intro` tactics are constructed from the rules given by the refiner. The `onSomeHypT` tactical searches the hypothesis of the goal sequent for a successful application of the `imp_elim` or `axiom` rules.

The implementation of the tactic primitives is shown in Table 9.2. The `prim_tactic` type is implemented as the refiner type `proof`, the `tactic_of_rule` function calls the refiner with the goal argument, and the `refine` function is just the identity. The `andthenT tac1 tac2` tactical performs the refinements specified by the tactics `tac1` and `tac2`, then composes the result. The `orelseT tac1 tac2` tactical applies `tac1`, catches the exception and applies `tac2` if `tac1` failed.

Table 9.2: Tactic module signature

```
module Tactics (Refiner) = struct
  type prim_tactic = logic → proof
  let tactic_of_rule rule t logic =
    Refiner.refine logic t
  let andthenT tac1 tac2 t logic =
    let p = tac1 t logic in
    let tl = Refiner.subgoals_of_proof p in
    let pl = List.map (fun t → tac2 t logic) tl in
      compose p pl
  let orelseT tac1 tac2 t logic =
    try tac1 t logic with
    RefineError → tac2 t logic
end
```

9.3 Multithreaded Refinement

Multithreaded refinement is useful in interactive proofs because it allows the user to run several tasks at once, making it possible to continue interactive development of unfinished proof nodes while other nodes are being searched by the refiner. In our implementation, multithreading is a prerequisite to distributed refinement. In this Section, we cover the main issues for multithreading: the implementation of a proof tree data type that supports concurrent operations, and a policy for thread scheduling.

9.3.1 Proof tree data type for concurrent processes

A tactic proof constructs a primitive proof tree, in which each node in the tree is labeled by a goal sentence and a tactic that was applied to the goal to produce the children. An example proof tree is shown in Figure 9.3. While the proof is being performed, the *frontier* of the tree contains pending subgoals. The diagram at the right shows a partial proof tree. The root is at the bottom and nodes labeled with `andthenT` tacticals are noted with an arc. In the Figure, the root is and-branching. The leftmost node has been explored first; it is or-branching where the leftmost child has failed, and the second child is and-branching. Since tactics are (by design) functional, the *order* in which the frontier nodes are explored is unimportant, and we can assign threads to multiple nodes in the frontier.

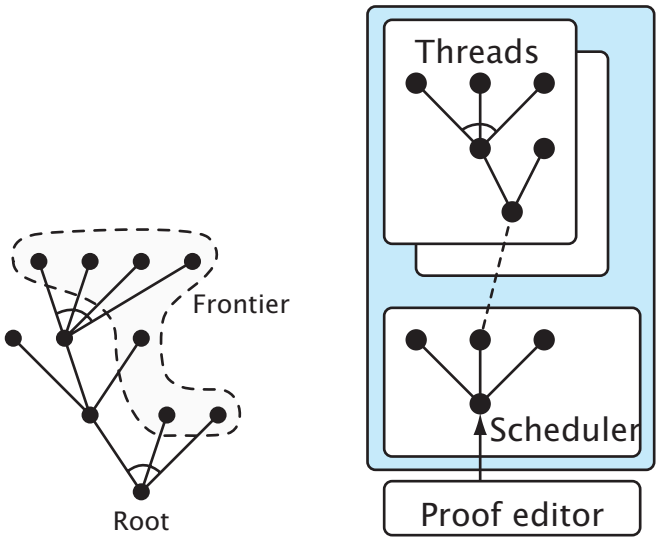

Figure 9.3: A partial proof tree and a proof scheduler

To implement the proof tree data type, we label nodes in a proof tree with seven kinds of labels:

```
AndThen1 (goal, tac1, tac2)
AndThen2 (tac2)
AndThen3
OrElse1 (goal, tac1, tac2)
OrElse2
Success (p)
Failure
```

The `AndThen1` and `OrElse1` labels correspond to applications of the tacticals at leaf nodes in the tree. The `AndThen2`, `AndThen3`, and `OrElse2` labels are used to label interior nodes as the proof tree is expanded. The `Success (p)` labels a leaf node for a successfully completed proof, and the `Failure` label marks frontier nodes that have raised an exception.

The complete set of operations on proof trees is shown graphically in Figure 9.4. These operations define expansion of `AndThen1` and `OrElse1` nodes, and they define how to back-propagate `Success` and `Failure` labels. A tree is complete when the root node is labeled either `Success(p)` for some proof `p`, or `Failure`. Note that two operations, *failure_andthen* and *success_orelse*, cause pruning of sibling nodes in the proof tree.

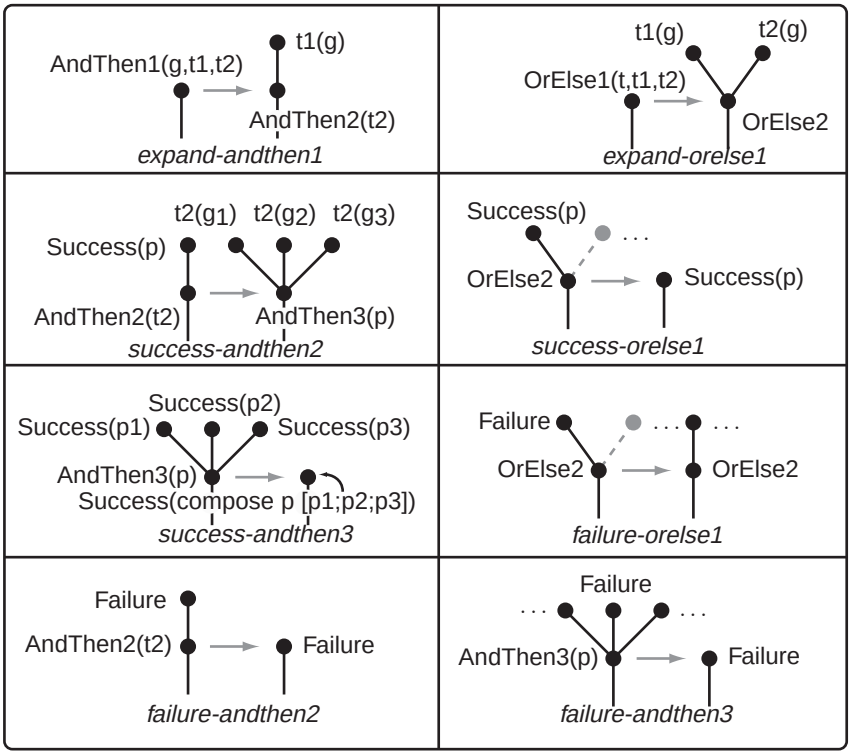

Figure 9.4: Proof tree operations

9.3.2 Threaded Refinement

To implement threaded refinement, we need to agree on a policy for the expansion of frontier nodes in the tree. The implementation we use in MetaPRL schedules the expansion into two components, shown in Figure 9.3.1. There is a centralized scheduler (the *Scheduler*) that manages the root of the proof tree, and there are multiple threads that schedule subtrees chosen by the Scheduler. The Scheduler does not perform node expansion directly; it passes unexpanded frontier nodes to idle threads, and handles responses from completed threads. The scheduling policy used by the Scheduler in MetaPRL is random.

The reason for this design is efficiency. Communication between threads is costly, and it is more likely that proof nodes near the root of the tree have large subtrees. Also, while the Scheduler uses a general scheduling algorithm policy, the individual threads use an efficient left-to-right depth-first search.

There are four operations for communication between the Scheduler and the threads, shown in Figure 9.5. The Scheduler submits a frontier node to an idle thread with the *submit* operation, and it receives the result from a completed thread with the *done* operation. When a branch of the tree is pruned, a thread may be terminated with the *cancel* op-

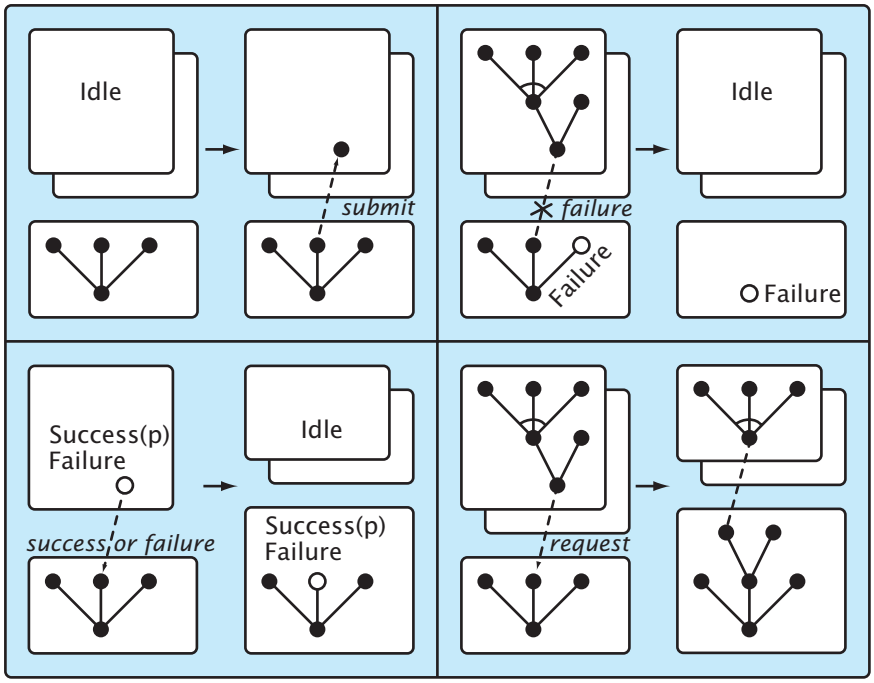

Figure 9.5: Scheduling operations

eration. If there are idle threads and no available frontier nodes to be scheduled, the scheduler can request that a thread return the root part of its tree to the scheduler with the *request* operation. By assumption, thread expansion is left-to-right, and the thread retains a single root node after the *request*.

Communication between threads is done by the OCaml `Event` module, defined with the following interface.

```
module Event :  sig
  type α event
  val create :  unit → α event
  val post :  α → α event → unit
  val select :  α event list → α
  val wrap :  α event → (α → β) → β event
end
```

An α `event` is a communication channel for messages of type α. The `post` function sends a message on the channel; the `select` function suspends the calling thread until a message arrives on one of the events in the list and the message is returned.

The thread interface to the scheduler has just three functions: one to `submit` a node to be expanded, another to `request` the root node from the running thread, and another to `cancel` the subtree expansion. The result of an expansion is returned through the event produced by the `submit` function; the `cancel` function causes `Failure` to be returned immediately.

```
module type ThreadSig = sig
  val submit :  node → node event
  val request :  unit → unit
  val cancel :  unit → unit
end
```

The Scheduler, shown in Table 9.3, implements the tactic interface as follows. A tactic produces an unexpanded proof node. The `tactic_of_rule` function provides the base case, returning `Success (p)` if the primitive refinement had proof `p`, `Failure` otherwise. The `andthenT` and `orelseT` tacticals perform no immediate computation; they produce proof nodes, which are scheduled and expanded by the `refine` function.

9.4 Distributed refinement

In distributed refinement, tactic evaluation occurs on multiple processors, and communication between refinement processes is through message passing. Our implementation of distributed refinement is symmetric; for each refiner process, we add an additional thread, called a *distribution service*, that requests frontier proof nodes from the Scheduler and passes them to remote refiners. There are two parts to the implementation: the *communication* model, and the *distribution service*.

Table 9.3: Scheduler implementation

```
module Scheduler (Refiner) = struct
  type node =
      Success of Refiner.proof
    | Failure
    | AndThen1 of term * tactic * tactic
    | AndThen2 of node * tactic
    | AndThen3 of Refiner.proof * node list
    | OrElse1 of term * tactic * tactic
    | OrElse2 of node * node
  and prim_tactic = node
  and tactic = term → prim_tactic
  let tactic_of_rule rule t logic =
    try Success (Refiner.refine logic rule t) with
    RefineError → Failure
  let andthenT tac1 tac2 t = AndThen1 (t, tac1, tac2)
  let orelseT tac1 tac2 t = OrElse1 (t, tac1, tac2)
  let refine t tac logic =
    match schedule (tac t) with
      Success p → p
      | Failure → raise RefineError
end
```

9.4.1 Distributed Memory Architecture

The communication model that we implement uses the Ensemble group communication system, in which processes form groups. Ensemble provides failure detection when processes can no longer communicate with the group, and it supports broadcast communication in the group with the additional guarantee that all process in the group receive messages in the same order (this is provided by the *total-ordering* protocol).

Ensemble implements an *upcall* interface, where an Ensemble application receives communication events through the following interface.

```
module type EnsembleUpcalls = sig
  type process, message
  val new_view :  process list → message list
  val heartbeat :  unit → message list
  val install_trigger :  (unit → unit) → unit
  val receive :  message list → message list
end
```

These *upcalls* are implemented by the application and passed to Ensemble during initialization. Most upcalls return a message list, which Ensemble broadcasts to the group. Messages are received through the `receive` upcall. Messages can be sent using the `heartbeat` upcall, which can be triggered by the function provided by the `install_trigger` upcall, which is called once when the application is initialized. The `new_view` upcall provides information about group membership. If the status of the group changes, the `new_view` function is called with a new list of processes belonging to the group.

We implement a shared-memory model using Ensemble. The memory entries initially correspond to frontier proof nodes. Entries may be *locked* or *unlocked*. Locks are exclusive; if an entry is locked, it is locked by exactly one process. Entries are created in the unlocked state, and the process that created the entry is called its *owner*. When a refiner process has no more unfinished frontier nodes, it may request a lock on an entry, with the intention of expanding it and returning the result value to the owner.

```
module DMA : sig
  type (α, β) handle
  val store :  α → (α, β) handle
  val lock :  ((α, β) handle → unit) → α handle
  val delete :  (α, β) handle → unit
  val return :  (α, β) handle → β → unit
  val event_of_handle :  (α, β) handle → β event
  val get_value :  (α, β) handle → α
end
```

Each process in the group keeps a local version of the shared memory, with entries containing the process that *created* the entry (with the `store` function), the process that *locked* the entry (with the `lock` function), and the value of the entry. The first argument to the `lock` function is a cancellation function to be called if the entry is *deleted* with the `delete` function. Only the owner of an entry is allowed to delete it (unless Ensemble determines that the owner failed, in which all copies of the entry are deleted). Once a `handle` to an entry is obtained, the value associated with the entry is be obtained with the `get_value` function. Once a computation finishes, a process *returns* a value to the owner with the `return` function, which deletes the entry from the memory.

The memory is implemented with Ensemble's broadcast mechanism. When an entry is created with the `store` function, the entry is broadcast to all processes in the group. When a lock is requested, an unlocked entry is selected from the local version of the memory, and a lock request is broadcast to the group. The lock request is *successful* if it is the first lock request to be received for that entry. Locks are granted only to the first requester; the total ordering property of Ensemble guarantees that all processes in the group agree on message ordering. If a lock request fails, and there is another unlocked entry in the shared memory, the request is issued for the new entry. Otherwise, the lock is postponed until a new entry is created.

When a `new_view` is initiated by Ensemble, it means the the group membership may have changed because of failures or the creation of new refiner processes. The `new_view` function delivers a new set of processes that compose the group. When a process fails, the entries it *created* are deleted, and its outstanding locks are removed. When a process joins the group, or when two groups merge, different process may have different local versions of the shared store. At the `new_view` a group coordinator is elected (the oldest surviving process in the group). Each process sends its local store to the coordinator, which merges the copies, deletes any entries for which the owner has failed, and broadcasts the result.

9.4.2 Distributed Refinement

Distributed refinement is implemented by augmenting the threaded refiner with a *distribution service* (DS). A block diagram of the distributed refiner is shown in Figure 9.6. To the Scheduler, the DS acts like a thread (it implements the `ThreadSig` interface). When the scheduler `submits` a node to the DS, the DS `stores` the node in the shared memory. If the Scheduler `cancels` the node, the DS `deletes` the entry from the shared memory. The Scheduler treats the `request` event specially. When the

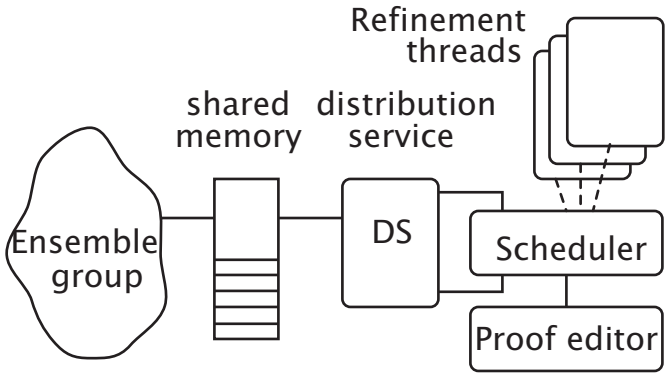

Figure 9.6: Distributed refiner block diagram

Scheduler makes a `request` for a new node, the DS attempts to `lock` an entry in the shared memory. If the lock is successful, the subgoal associated with the entry is passed to the Scheduler as a new frontier node. When the Scheduler completes the evaluation, it passes the result back to the DS, which `returns` the result to the owner.

In the Figure, the proof editor is optional; there may be only one process in the group with a proof editor. When an idle process joins the group, it synchronizes its shared memory, and it may pass new nodes to the scheduler for refinement. As the scheduler runs, new subgoals are passed back to the DS to be performed remotely. As a large proof is developed, a complex dependency graph is established between the proof nodes of different process. When a process fails, a large portion of the proof tree may be pruned.

9.5 Performance Measurements

We include performance measurements for four domains:

- The `Pigeon` domain is a proof of the propositional pigeonhole principle. The `Pigeon3` problem shows that 4 pigeons do not fit into 3 holes, and the `Pigeon4` problem shows that that 5 pigeons do not fit into 4 holes.
- The `Gen` domain is a proof of heredity in a large geneological database in first-order logic.
- The `Term` domain is a proof of correctness of `term` operations in the Nuprl type theory.
- The `CZF` domain is a proof of the derivation of Aczel's CZF set theory from the Nuprl type theory.

Problem	unthreaded	1 proc	2 proc	3 proc	4 proc	5 proc
Pigeon3	7	8.6	7.2	6.2	4.5	3.8
Pigeon4	207	221	141	95	80	67
Gen	109	118	73	41	28	20
Term	422	440	250	201	160	137
CZF	63	75	40	32	26	22

Figure 9.7: Performance on the five domains

The Pigeon and Gen domains use fully automated proof search. The Term and Set domains are automated replays of proof transcripts for interactively generated proofs. The transcripts contain the full search problem posed by the user; the Term domain contains about 2700 interactive proof nodes, and the CZF domain contains about 2300 (the proof steps in CZF are significantly simpler than those in Term). These two domains each represent about three weeks of interactive development.

The runtimes for each of these problems are shown in Figure 9.7 for one to five processors in the Ensemble group, running on Linux 200MHz Pentium 686 machines. We also include data points for the original unthreaded tactic primitives for comparison.

The use of the distributed prover usually incurs a penalty over the the unthreaded prover. The speedup on these problems is fairly consistent, averaging about 3.2 with for the 5 processor case. The GEN problem is an exception; it acheived a superlinear speedup. On this problem, the random scheduling algorithm performs better than the default depth-first-search.

The communication cost on these problems is high—the largest GEN problem generates 0.45MB/sec in communication. The proof nodes that are being sent between machines contain tactics and proof trees that grow to several 100KB as the problem runs. Much of this communication is unnecessary, since the proof trees are often copied and discarded. Total memory requirements also increase with the cost of communication because the OCaml marshaler makes a deep copy of every value that is sent on the network.

Communication cost tends to be the major bottleneck. On these problems, network latency was a greater problem than lock contention. In addition, the larger problems were more likely to undergo false failures. A generic technique for addressing these problems would be message compression: most goals that are passed sequentially across the network share common subterms. A caching marshaler would preserve sharing for common subterms in multiple message, addressing

the problem of message size and memory usage.

9.6 Summary, implementation notes, and future work

The distributed architecture we have presented owes its simplicity to the modular design of MetaPRL and Ensemble. The distributed tactic module in MetaPRL transparently replaces the unthreaded tactic interface between the tactic library and the logic engine, making it unnecessary to modify the tactic library. Ensemble provides a concise interface—although the mechanism for ensuring consistency is quite complex.

Several interesting problems arose in the implementation. One problem was that Ensemble was not designed for a multithreaded environment, and it initially provided poor performance. When our effort to augment Ensemble achieved only modest gains, we separated the MetaPRL and Ensemble processes. Communication between MetaPRL and Ensemble is currently is through (physical) shared memory.

OCaml provided essential communication support for communicating tactics as *functions*. The OCaml marshaler provides native support for marshaling arbitrary OCaml values (including functions) to byte streams. This is not without its pitfalls. From the programming perspective, it is difficult to know which values are part of a function's closure, and the MetaPRL implementation initially attempted to marshal the entire Ensemble data structure into every message! The solution was careful coding to avoid spurious bound variables in the tactic functions. Communication becomes a bottleneck as group sizes increase, and a more sophisticated marshaler would reduce message sizes by caching recurring values.

The speedups we obtain are due to the wealth of parallelism in the proofs. However, efficiency can be greatly increased in some domain with domain-specific scheduling policies. Another issue is proof caching, where commonly occurring subgoals can be identified. We leave these two issues as future work.

9.7 Related Work

There are several examples of first-order proving and limited higher-order distributed theorem provers [BM94, SP92, CV98, DF97]. These efforts differ from ours in at least two respects: we focus on general-

purpose provers that use a mixture of interactive and automated proof, and we require fault-tolerance. However, the scheduling problem is a common issue in all implementations, and the insight gained from first-order provers may lead to a more general scheduling policy mechanism in our framework.

Our communication model (the DMA) is quite similar to Leiserson's CILK system [FLR98], in which parallel and distributed programs are written using annotated C code. Program distribution shares many of the mechanisms we describe for the Ensemble shared memory. An account of Ensemble can be found in Hayden [Hay97, Hay98].

Part IV

Theoretical Issues and Extensions

Theoretical issues

This Part of the dissertation is concerned with theoretical concerns and outstanding issues, primarily having to do with the *module* system. The module system is currently not fully formalized. Although we can derive specific relations between theories, we can't quantify over the class of *all* computable theory relations. In essence, the theory relations are not first-class.

Chapter 10 presents an attempt to classify, and thus formalize, relations between theories. The chapter also presents the semantics of the "very-dependent function type" described in the Nuprl module `Itt_rfun` (Section 14.11). While the semantic basis for the very-dependent function type is well-grounded, the semantics of higher-order theories is less so, and the thoughts presented in the Chapter may be interpreted as one possible foundation.

The ideas for formalizing higher-order modules are closely related to semantic interpretations of classes, objects, and inheritance. Chapter 11 presents a formalization of objects in the Nuprl type theory. The result gives a quite satisfying interpretation of object-oriented programming, where classes are interpreted as records of polymorphic methods, and objects as records of *monomorphic* methods.

Chapter 10

Modular Theories, and the "Very-Dependent" Function Type

Constructive type theories are powerful systems for describing mathematical objects with complex dependencies between types and computational values, making them expressive enough to encompass large ares of mathematics and programming. However, as the body of formal knowledge in the type theory expands, the problem of managing mathematical domains and their proofs becomes increasingly significant. Though each of the objects in a theory are formal, the theory itself is not. In this chapter, we show how to apply the methods of formal data abstraction to the organization of the mathematical domains. In the process we expand the tools of data abstraction to include reusability and namespace organization, providing an environment that can be used for defining objects within a domain, for organizing domains within a type theory, and for organizing theories within a system. This environment will require extending the expressivness of inductive definitions within the type theory to include the dependent characteristics of type theoretical domains.

10.1 Introduction

In a practical sense, modern constructive type theories are undergoing a crisis of scale. While it is possible to formalize significant areas of mathematics and programming, in most cases the effort involved is still much greater than is practical. Yet the application of formal methods to large practical problems continues to hold a great deal of appeal: as programs get larger, they become more difficult to reason about, making automated assistance even more desirable. These contradicting goals make it tempting to use type theories that favor practical techniques

over semantical meaning, or that abandon formal methods entirely and rely on informal proof procedures.

Of course, informal methods and heuristics will always be a part of automated verification and synthesis (to help deal with the fact that provability is undecidable). However, the verification problem eventually repeats itself; a verification procedure that does not catch errors is not very useful, and a procedure that signals false errors is often worse. If we could prove that the verification procedure is correct, the payoff would be big.

The problem of scale in type theories has two aspects. First, formal proofs are often much larger and more difficult than proofs on paper because the formal system has no idea of what is "obvious." Second, the type theories are intrinsically not scalable; even though the objects of the type theory are formal, their organization is *ad-hoc*. We don't address the first issue in this chapter. It is a difficult problem, and it may only be alleviated by finding more powerful algorithms for proof search. The second issue is more tractable. There is a great deal of experience, both formal and informal, on using organizational tools to deal with the problems of scale. From abstract algebra, we have formal definitions for modularity and abstract data types. From the classification methods of taxonomy, we get an orthogonal organization scheme that we can use to provide abstract data types with inheritance and re-use (making them "object-oriented"). Finally, from software engineering, we can use experience in naming to classify and identify the objects we create.

Before we proceed, it is useful to provide some terminology. We use the word "object" to refer to data that belongs to a "signature," consisting of a collection of named "methods." The methods each have an arbitrary type, and the signature is the product of those types. Signatures are themselves types, and may be used to specify the method types of further objects. We differ from the standard (for example Abadi and Cardelli [AC94b]) in that the methods are simple types, not dependent upon the signature being defined. By using this loose definition, we will be able to apply our treatment to each part of the type theory, including the type theory itself and the *theories* (or *mathematical domains*) that are defined within it. We also make this restriction so that we can provide a well-defined semantic meaning to the type of a signature that will not rely on a fixed point construction.

The type that we will assign to the signature of an object will be a dependent record type. That is, a signature will consist of a collection of labeled types that depend on one another in some well-founded way. As it happens, this will require that we extend the type theory with a suitable "very-dependent" type, which is an interesting type-theoretical problem in itself. In the following sections of the chapter, we begin

with an example that illustrates the features we will need from the type theory. Following that, we describe a very-dependent type extension to the Nuprl type theory, and we cover parts of the type theory to give some intuition about the extension. In Section 10.4, we develop the objects and give an informal account of an "object system." In Section 10.5, we extend the object system to include specifications of mathematical domains.

10.2 Module example

Before proposing an extension to the type theory, it will be useful to look at an example of a module specification that we would like to describe. This will point out some of the features that we will want from the type theory.

An obvious example comes from abstract algebra (we would hope that algebraic objects have easy descriptions). One of the first objects studied in an elementary algebra course is the semi-group, or monoid. The description is roughly as follows: a monoid is a triple $(S, \oplus, e)$, where S is a set called the *carrier*, $\oplus: S \times S \to S$ is a binary operation on the set, and $e \in S$ is an identity of $\oplus$. Since we will be giving a formal account in a constructive type theory, the carrier will be specified by a type *car* and a decidable equality = (the type denoting the carrier will, in general, have a coarser equality than the set we are interested in). Within the type theory, the monoid can be specified with the following type:

$$
\begin{array}{lll}
\text{Monoid} & == & \mathit{car}: \mathit{Type} \\
& \times & =: \mathit{car} \to \mathit{car} \to \mathit{Prop} \\
& \times & \mathit{eq\text{-}prop}_{\mathit{ref}}: \forall x: \mathit{car}.\, x = x \\
& \times & \mathit{eq\text{-}prop}_{\mathit{sym}}: \forall x, y: \mathit{car}.\, x = y \Rightarrow y = x \\
& \times & \mathit{eq\text{-}prop}_{\mathit{trans}}: \forall x, y, z: \mathit{car}.\, x = y \Rightarrow y = z \Rightarrow x = z \\
& \times & \oplus: \mathit{car} \to \mathit{car} \to \mathit{car} \\
& \times & \mathit{assoc\text{-}prop}: \forall x, y, z: \mathit{car}.\, (x \oplus y) \oplus z = x \oplus (y \oplus z) \\
& \times & e: \mathit{car} \\
& \times & \mathit{identity\text{-}prop}: \forall x: \mathit{car}.\, x \oplus e = x
\end{array}
$$

The dependencies can be determined syntactically: every method depends on *car*, except *car* itself. The eq–prop$_*$ methods depend on = but not on each other, etc. This particular signature can be expressed as 9–ary dependent cartesian product. The signature itself is just a type. If we wish to specify a group, we can just extend the signature of the

monoid by adding the axiom that a group must have an inverse:

Group **is** Semi-group $\times$ *inverse–prop*: $\forall x$: *car*.$\exists y$: *car*.$x \oplus y = 1$

In this case, the signature of the group is a binary product, containing the signature for the monoid, plus the *identity* axiom.

There are at least three problems that are immediately apparent:

1. There is no consistent signature type. Signatures are just types, which may be binary products, 9-ary products, or any other type. This affords considerable flexibility but little structure.

2. Coercions must be explicit. We would like to say that a group *is* a monoid, without having to provide the explicit coercion function. Put another way, we would like groups to be a subtype of monoid. And yet another way: we would like all theorems about groups to apply to monoids transparently.

3. There is no support for multiple inheritance (in fact there is no support for inheritance). The methods of the monoid are not directly available in the group. Furthermore, if we describe a module with two parents, the common ancestors won't be joined.

We can solve these problems by introducing a level of indirection. That is, we introduce a type of *descriptions* of signatures. This could simply be a list of labels and types assigned to the labels. We would like to have a type of all signature descriptions, so this would be a recursive type. Furthermore, the types associated with each label depend on inhabitants of previous types in the list. Thus, the ideal type for signature descriptions would be a dependent record. For instance, the signature description for the monoid would be the record:

Monoid–description == [	
car,	*Type*;
=,	$car \rightarrow car \rightarrow Prop$;
$eq\text{–}prop_{ref}$,	$\forall x$: *car*.$x = x$;
$eq\text{–}prop_{sym}$,	$\forall x, y$: *car*.$x = y \Rightarrow y = x$;
$eq\text{–}prop_{trans}$,	$\forall x, y, z$: *car*.$x = y \Rightarrow y = z \Rightarrow x = z$;
$\oplus$,	$car \rightarrow car \rightarrow car$;
assoc,	$\forall x, y, z$: *car*.$(x \oplus y) \oplus z = x \oplus (y \oplus z)$;
e,	*car*;
identity,	$\forall x$: *car*.$x \oplus e = x$
]	

The signature of the monoid is a dependent record type corresponding to this description, and monoids are dependent records inhabiting the signature. By constraining signatures in this way, we can define a generic set of operations that are well-formed on all objects having a signature, solving the first problem. The dependent records use transparent record subtyping. So, for instance, a group is a monoid, because it contains all the fields of a monoid. The coercion is implicit, because it just requires "forgetting" the extra field of the group, solving the second problem. We can solve the last problem by designing the signature generation algorithm so that it merges common ancestors. Using the dependent record type, method selection corresponds to field selection. Since the group is a monoid, the methods of the monoid are inherited by the group. The problem of multiple inheritance is not completely solved: if we consider rings (which contain a monoid and a group with a common carrier), then a ring can be coerced to a monoid in two distinct ways. We will discuss this issue in section .

Since the type theory doesn't provide a constructor for dependent records, that will be the first extension we will make. In the next section we generalize the goals of dependent records to a class of type we call "very-dependent" types. These types have the common feature of potentially describing an infinite number of dependencies.

10.3 Very-dependent type extension

Most constructive type theories have types that can express complex dependencies between types and values, or between types and types. These dependencies are crucial for providing precise specifications for computational objects. For example, the base type theory generally provides a dependent function type, often written as $x{:}A \rightarrow B$ or $[x{:}A]B$, where A is a type, and if x is an object inhabiting A, then B is a type. The type A describes the domain of the function, and $B(x)$ describes the range of the function on argument x. The dependent type can be used to provide a more precise specification for a set of functions than the undependent function type. For instance, using the dependent type we can define a class of natural number functions whose output is always larger than their input with the type $F \equiv i{:}\mathbb{N} \rightarrow \{(i+1)\ldots\}$. While each function described by F is also included in $\mathbb{N} \rightarrow \mathbb{N}$, clearly F is a more precise specification for the set of functions we are interested in.

Inductive types are used to describe recursive objects such as lists and trees. In Nuprl [CAB+86] simple inductive types are described with an inductive type constructor $\mu(X.T)$ where X is bound in the term T and represents the fixed point, i.e. $\mu(X.T) = T[\mu(X.T)/X]$. For this

type to be meaningful according to its semantics, the term T must be a type that is monotonic in X for *any* type X. In Coq [PM93], a similar construction with a *syntactic* constraint can be used to define inductive types for the decidable type theory. Similarly, HOL [GM93] contains a mechanism for introducing primitive recursive types.

The inductive types are useful for a variety of purposes. They are used to define inductively defined predicates, relations, and also to define abstract data types for some degree of modularity. Most mathematical reasoning requires use of induction, and inductively defined types are fundamental to most modern computation systems. Driven by the essential importance of inductive types, some proof systems, notably it Coq, code most type constructors on top of the inductive type, so that the only type constructions needed are the inductive type and function space.

All of the forms of inductive types are used to describe recursive objects *without* type dependencies over the recursion. For example, in Nuprl, given a type T, we can describe a list of objects of type T with the type $\mu\,(X.Unit + \prod TX)$, where *Unit* is a type inhabited by a single object. The objects in the list each have type T, and they are each independently described. If T were the set of integers, the list would contain integers having no relation to one another.

Now suppose we wish to describe recursive objects where there are dependencies over the recursion: for instance, lists of sorted integers. The more general form of the inductive type constructor in Nuprl allows a parameter to be passed to the construction, along with an initial value for the parameter. Parameterized types are defined with the construction $\mu\,(X, i.T; v)$, where X and i are bound in T, X represents the parameterized type fixed point, i represents the parameter value, T is the type construction, and v is the initial value of the parameter. Using the parameterized recursive type, we can describe increasing lists of natural numbers with the type

$$L_I \equiv \mu\,(i, X.Unit + j{:}\{i\ldots\} \times X\,(j+1)\,;0)\,.$$

Note that the dependencies are from the front of the list: the first element must be 0 or larger, the second must be larger than the first, and so on. We can define a constructor for this list as *cons*, where

$$cons(h,t) \equiv inr\,\langle h,t\rangle$$

This constructor creates objects of type L_I if, and only if, t is an list of increasing integers and it is empty, or its first element is larger than h. Thus, using this construction the second argument to *cons* must belong to a subtype of L_I that depends on the first argument.

label	type
x_0	f_0
x_1	$f_1(x_0)$
x_2	$f_2(x_0)(x_1)$
x_3	$f_3(x_0)(x_1)(x_2)$
$\vdots$	$\vdots$

Figure 10.1: Telescope

Although the typing rule for *cons* is somewhat awkward, it is sufficiently practical that is can be used for normal program development. However, consider the type of *telescopes* as described by DeBruijn [DB]. For our purposes, a telescope is a vector of objects as shown in Figure 10.1. The type of element x_i in the vector is given by a function f_i that produces a type given the first $i-1$ elements of the vector. That is, the first element x_0 of the vector has type f_0, the second element x_1 has type $f_1(x_0)$, the third $f_2(x_0)(x_1)$, and so on. If we are to give a type for the telescope we have two problems: what is the type of the type-function vector $\mathbf{f} = f_0, f_1, \ldots, f_n$, and what is the type of the telescope $\mathbf{x} = x_0, x_1, \ldots, x_n$?

Assuming that we can give a type for the type-function vector, and we have an actual type-function vector $\mathbf{f}$, the type of the telescope can be described with the parameterized recursive type

$$T \equiv \mu\,(i, X.\mathit{Unit} + x{:}\,(\mathit{nth}(\mathbf{f}, i.1)(i.2)) \times X((i.1)+1, \langle i.2, x\rangle); \langle 0, \cdot\rangle)$$

Here, the function $[\mathit{nth}{:}\prod \star\ \mathit{list}\mathbb{N} \to \star\,]$ computes the n^{th} element of a list, and i has the informal type $j{:}\,\mathbb{N} \times \prod f_0 \prod f_1 \prod \cdots f_j$.

There is no *cons* operation for a telescope, because it doesn't make sense to add an element to the telescope before x_0. Nor is it possible to give a type for telescopes represented in reverse order (at least using the parameterized recursive type). The dependencies between types and values in the telescope must proceed from the first element to the last, and telescopes must be constructed using an *append* operation.

The "direction" of the dependencies ultimately limits the expressivity of the parameterized recursive type. Suppose we wish to generalize the linear telescope to trees of values, where the type of the value of each vertex depends on the values of all the children, as shown in Figure 10.2. Using the parameterized recursive type it would be difficult to describe the tree because the description would necessarily begin at the leaves. Any such description would be so unnatural as to be prac-

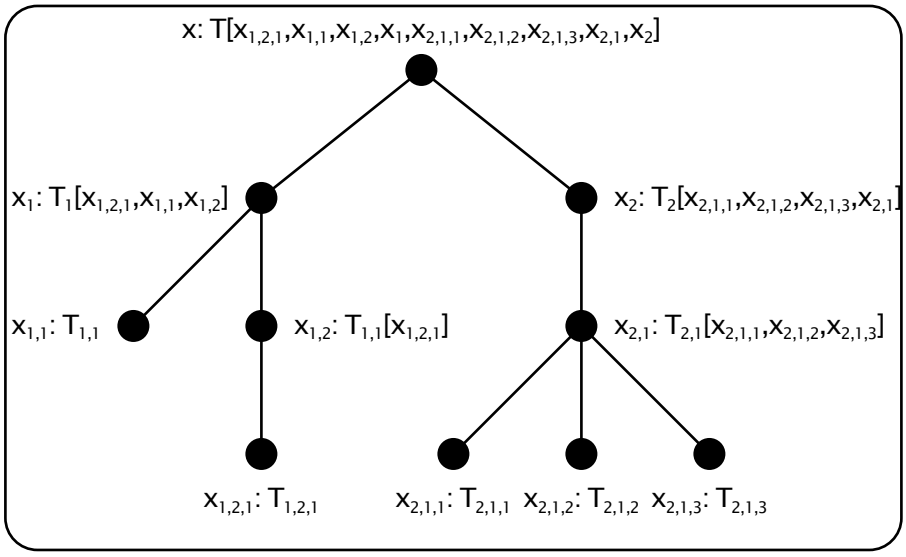

Figure 10.2: Tree of dependent types

tically unusable. The natural solution is to provide a type constructor that would allow the tree to be described from the root. Assuming these trees have type T, the root has children of type T, and it also has a type that depends on the specific children. We use this as the definition of a *very-dependent* type: that an object of the very-dependent type T contains subobjects that have type *dependent* upon proper subobjects of type T.

At first glance, it seems that we can define very-dependent types using the obvious dependent construction. For instance, to describe the list of increasing natural numbers, we could give the type:

$$L'_I \equiv \mu\,(X.Unit + l{:}X \times (case(l)\ of\ inl\ _ \Rightarrow \mathbb{N} \mid inr\ \langle i, _\rangle \Rightarrow \{\ldots i\}))$$

This type specifies that the *last* element of a list must be a natural number, and all other elements must be *smaller* than the next successive element. The *cons* constructor is the same as for the previous integer list, but now $cons(h, t)$ is well-defined if t inhabits L'_I and h is a natural number that is smaller then than the first element of t if t is not empty.

The problem with this construction is that the semantics of the μ recursive type require that the term $Unit + l{:}X \times (case(l)\ of\ inl\ _ \Rightarrow \mathbb{N} \mid + \Rightarrow inr\ \langle i, _\rangle\{\ldots i\})$ denote a type for *any* type X. But this clearly doesn't hold since the *case* expression is only defined if l belongs to a disjoint union.

It is critical here to understand that the *semantics* of the μ inductive type forbids very-dependent type constructions. The reason is that the fixed point is not bounded, except over *Type* (the collection of all

types). In order for the fixed point to be well-defined, the definition must be well–formed for any value of the type variable X. This forbids dependent constructions because they require some structure on the dependen type. If we are to propose a solution, we will need to provide a different construction with its own semantics. Indeed, as we will show in section 10.3.2, we can give a different semantics for the μ construction that allows the type construction above.

In the next section, we explore the first extension to the type theory, the type of *very-dependent* functions. Although we don't normally think of function types as being recursive, we can see the correspondence by considering functions with domain $\mathbb{N}$. These functions are isomorphic to infinite lists, and the dependencies in the type of the elements of the list corresponds to having a range type for a value depend on "previous" values of the function. This concept can be extended to general well-founded domains.

10.3.1 Very-dependent function type

The normal dependent function type allows dependencies to be expressed between the argument of the function, and its range type. For instance, the Nuprl type $i{:}\mathbb{Z} \rightarrow \{i \ldots\}$ is the type of functions taking integer arguments and returning integer values that are at least as large as their argument. However, there are no dependencies between the types of values of the function on different arguments.

In order to introduce such dependencies, we introduce a new type constructor $\{f \mid x{:}A \rightarrow B[f,x]\}$, where f and x are bound in B. This corresponds to a function f with domain type A, and range $B[f,x]$ on argument $x \in A$.

Example of a very-dependent function

Before proceeding with a more formal treatment, it may be useful to present an example. Suppose we wish to encode an n-ary dependent product $x_1{:}T_1 \times x_2{:}T_2[x_1] \times \cdots \times x_n{:}T_n[x_1, \ldots, x_{n-1}]$. We can encode this type using the very-dependent function type, where the inhabitants are functions with domain $D = \{0 \ldots (n-1)\}$, using the type

$$T_f(n) \equiv \left\{ f \;\middle|\; i{:}D \rightarrow \left[\begin{array}{lcl} \mathit{case}(i)\ \mathit{of} & & \\ 0 & \Rightarrow & T_0 \\ 1 & \Rightarrow & T_1(f\,(0)) \\ & \vdots & \\ n-1 & \Rightarrow & T_{n-1}(f\,(0))(f\,(1)) \cdots (f\,(n-2)) \end{array} \right] \right\}$$

If we have a function F that returns the values T_i for $0 \le i < n$ given the index i, we can give this type a more concise form. Let $\mathit{fix}(f.b)$ define the fixed point of the program b. That is, $\mathit{fix}(f.b) \equiv Y\,\lambda f.b$. Given a functions F and f, and an index i we can define a function *napply* that applies F to $f(0), \ldots, f(n-1)$:

$$\mathit{napply} \equiv \lambda f.\lambda i.\, \mathit{fix}(g.\lambda G.\lambda j.\, \textbf{if } j = i \textbf{ then } G \textbf{ else } g\,(G\,(f\,(j)))\,(j+1))$$

The arguments to *napply* are as follows:

i is an integer that is the index of the final argument,
j is an integer that is the index of the first argument,
G is the value that will be applied to the arguments $f(j), f(j+1), \ldots, f(i-1)$,
f is a function returning the arguments.

Computing *napply* gives

$$\mathit{napply}\,(f)\,(i)\,(G)\,(j) \equiv G(f(j))(f(j+1)) \cdots (f(i-1))$$

Using *napply*, the type $T_f(n)$ is then described as:

$$T_f(n) \equiv \{f \mid i{:}\{0 \ldots (n-1)\} \rightarrow \mathit{napply}\,(f)\,(i)\,(F\,(i))\,(0)\}$$

The type function F can be defined in a similar manner. Given F, and an index i, we can compute the i-ary dependent function type with the term:

$$\begin{array}{l}\mathit{nfun} \equiv\ \lambda F.\lambda i.\, \mathit{fix}(g.\lambda h.\lambda j.\\ \quad \textbf{if } j = i \textbf{ then}\\ \qquad \mathbb{U}_i\\ \quad \textbf{else}\\ \qquad x{:}\,\mathit{napply}\,(h)\,(j)\,(F\,(j))\,(0) \rightarrow\\ \qquad g(\lambda k.\, \textbf{if } k = j \textbf{ then } x \textbf{ else } h\,(k))(j+1)\end{array}$$

The arguments are:

i is an integer that is the index of the last type in the function type being constructed,
j is an integer that is the index of the first type in the functin type being constructed,
F is the value that will be applied to get the argument type of the function type being constructed,
h is a function defined on $\{0 \ldots (j-1)\}$ that returns the initial arguments of the dependent function type being constructed.

After computation, we have

$$
\begin{array}{ll}
& \mathit{nfun}\,(F)\,(i)\,(h)\,(j) \\
\equiv & x_j{:}F(j)(h(0))(h(1))\cdots(h(j-1)) \\
\to & x_{j+1}{:}F(j+1)(h(0))(h(1))\cdots(h(j-1))(x_j) \\
\vdots & \\
\to & x_{i-1}{:}F(i-1)(h(0))(h(1))\cdots(h(j-1))(x_j)\cdots(x_{i-2}) \\
\to & \mathbb{U}_i
\end{array}
$$

In the normal case, $j = 0$ and the argument h is not used, and we have

$$
\begin{array}{lll}
\mathit{nfun}\,(F)\,(i)\,(h)\,(0) & \equiv & x_0{:}F(0) \\
& \to & x_1{:}F(1)(x_0) \\
& \vdots & \\
& \to & x_{i-1}{:}F(i-1)(x_0)(x_1)\cdots(x_{i-2}) \\
& \to & \mathbb{U}_i
\end{array}
$$

The type for F is then:

$$T_F(n) \equiv \{F \mid i{:}\{0\ldots(n-1)\} \to \mathit{nfun}\,(F)\,(i)\,(\lambda x.x)\,(0)\}$$

If we parameterize these types over n, we have the types

$$
\begin{array}{lll}
S_f & \equiv & n{:}\mathbb{N} \times T_f(n) \\
S_F & \equiv & n{:}\mathbb{N} \times T_F(n)
\end{array}
\tag{10.1}
$$

The type S_F specifies *descriptions* of dependent products of arbitrary finite arity, and the type S_f specifies the corresponding dependent product type. The inhabitants of S_f are pairs $\langle n, f\rangle$ where n is the arity of the product, and f is a function with domain $\{0\ldots(n-1)\}$ that serves as the projection function.

This finite arity dependent product type will be useful to define the type *Signature*. It can also be used to define the type of sequents in a constructive logic: the hypotheses of the sequents are defined as a dependent product of finite arity, and the goal is a type that may depend on all hypotheses. From here we will cover a more formal treatment of the function type, beginning with its semantics.

Semantics of the very-dependent function type

The central issue in the semantics for the very-dependent function type $\{f \mid x{:}A \to B[f, x]\}$ is deciding what it means to use the function in the definition of its range. Is it possible to define the type of a function an an

argument in terms of the function's value on the argument? The immediate inclination is to answer no—consider the term $\{f \mid i{:}\mathbb{Z} \to f\,(i)\}$. If this were a type, it would violate predicativity because, for instance, $f\,(0) \in f\,(0)$. Would this type, however, make sense in an impredicative system with some *Type* where $\mathit{Type} \in \mathit{Type}$?

Even if we rule out impredicative types, we might want to allow some cyclic definitions. For instance, suppose we have some term that tests for computational convergence. Let $\{p\}\downarrow$ denote the fact that the program p converges. Consider the type T where,

$$T \equiv \{f \mid i{:}\mathbb{Z} \to \textbf{if } \{f\,(i)\}\downarrow \textbf{ then } \mathbb{Z} \textbf{ else } \mathit{Void}\}$$

This type might denote the partial functions with domain and range $\mathbb{Z}$.

In the next section, we will discuss a semantics for the very-dependent function type, restricted to non-cyclic definitions. There are interesting variations on the semantics that allow cyclic definitions, or definitions that derive the order from the computation in the range, or that allow the expression of partial function types. However, these semantics are an area of research—they do not fit in well with the current semantics of the type theory, and it seems likely that the semantics of the entire type theory would have to be modified to accomodate them. We don't discuss the extended semantics in this chapter.

Simple semantics The simplest interpretation for the very-dependent function type $\{f \mid x{:}A \to B\}$ requires that there be no cycles in the type definition. That is, the term $B[f, i]$ must not evaluate f on i for any $i \in A$, and it must have a normal form. The simplest way to enforce this property is to require that A be well-ordered according to some relation $<{:}A \to A \to \mathbb{U}_i$, and to require that $B[f, i]$ be well-defined for f with domain $\{a{:}A \mid a < i\}$. This is a conservative restriction, since only the subtype of A that is used in defining the range need be well-ordered.

With this restriction, we can give a semantics for the type $\{f \mid x{:}A \to B\}$as follows.

Semantics 1 *The term $\{f \mid x{:}A \to B\}$ is a type with membership φ, if and only if:*

1. *A is a type with membership α,*
2. *A is well-founded with respect to some partial order $<$,*
3. *for any a such that $\alpha(a)$, there is a $\gamma_{a'}$ for all a' where $\alpha(a')$ and $a' < a$ and the following hold:*
   (a) *$\{f \mid x{:}\{a''{:}A \mid a'' < a'\} \to B\}$ is a type with membership $\gamma_{a'}$,*

(b) $B[g; a']$ *is a type with membership* β_g *for any* g *where* $\gamma_{a'}(g)$,

(c) for any f, $\varphi(f)$ *if and only if, for all* a *where* $\alpha(a)$ *holds,* $\gamma_a(f)$ *and* $\beta_f(f(a))$ *also hold.*

This semantics is well-founded because A is well-founded. The membership predicates γ and β can be constructed by induction. In the base case, the type $\{f \mid x{:}\{a''{:}A \mid a'' < a'\} \to B\}$ reduces to $\{f \mid x{:}\mathit{Void} \to B\}$ which is a type for any term B. Similarly, the semantics requires that at the base case require that $B[g, a']$ be a type for the function g with domain *Void*, which means that B canot evaluate g on any argument. The definitions for γ and β are constructed mutually recursively for the entire domain A. We can summarize these points in the following theorem.

Theorem 6 *Semantics 1 are well-defined.*

Proof The semantic definition is well-founded, because the membership constructions are defined by induction, based on the well-ordering of A.

We must show that the membership function is well-defined. Suppose we have a type $\{f \mid x{:}A \to B\}$ that is defined according to Semantics 1, and consider an element f where $\varphi(f)$. By item 3a, we know that $\gamma_a(f)$ and $\beta_f(f(a))$ for any $a \in A$. It follows that f is a function. Any element g, where $g = f$, must also be a function with domain A. By extensionality, $g(a) = f(a)$ for any element $a \in A$, and so $\beta_g(g(a))$ and $\gamma_a(g)$ hold. It follows that $g = f \in \{f \mid x{:}A \to B\}$. □

We must now determine rules for the type that reflect the semantics. The most difficult rule is for type membership. For a type $\{f \mid x{:}\{a''{:}A \mid a'' < a'\} \to B\}$ to be well-formed, the requirements are:

1. A must be a type,
2. A must be well-ordered according to some relation $<$,
3. B must be a type defined by induction on the well-ordered prefixes of A.

The fact that A is well-ordered is necessary to fit the semantic characterization, but the computational content of the proof of well-ordering is irrelevant. In the rule, we can discard the computational content of the proof. The type A is well-ordered iff it has an induction principle. We state this as a proposition:

$$\begin{aligned}
&\mathit{WellFounded}_i\,(A; <) \equiv \\
&\quad \forall P{:}(A \to \mathbb{U}_i).\forall a_1{:}A. \\
&\qquad (\forall a_2{:}A.a_2 < a_1 \Rightarrow P\,(a_2)) \Rightarrow P\,(a_1) \\
&\qquad \Rightarrow \forall a{:}A.P\,(a)
\end{aligned}$$

The proof of well-foundedness is then stated with the "set" type, in order to discard the proof:

$$\downarrow \{\mathit{WellFounded}_i\,(A; <)\} \equiv \{\mathit{Unit} \mid \mathit{WellFounded}_i\,(A; <)\}$$

The inhabitant "$\cdot$" exists only if $\mathit{WellFounded}_i\,(A; <)$ is true, but the proof of $\mathit{WellFounded}_i\,(A; <)$ is not actually saved.

The rules can be developed along these lines to give the rules defined in the `Itt_rfun` module (Section 14.11). The equality rule enforces the requirements of the previous paragraph. There is no formation rule, because the recursion in the type definition requires that the entire type be constructed at once. For the extensional equality, the two terms being compared must be functions, they must return equal values on equal arguments, and the term of the equality must be a very dependent function type.

The rule for *rfunctionLambdaFormation* is interesting because its proof is by induction, and so the extract requires an induction combinator. The termination is enforced by the well-foundedness of the domain. Interpreted as a proposition, the type $\{f \mid x{:}A \to B\}$ is a recursive universal proposition. That is, if it is true, then for all $x \in A$, the proposition B holds *assuming* that the universal implication is true on smaller values of the argument. In essence, this *is* an argument by induction, and the recursive proof extract makes sense.

Note that it is not necessary to include the well-order of the domain in the very-dependent function type. That well-ordering is property that is used for the well-formedness argument where it is used to show that the definition is not cyclic; it is useless anywhere else. Also, the specific well-order is not important. There may be more than one well-order for which the type is well-formed. If so, there is more than one termination argument, but the termination holds in any case.

As we stated before, the semantics of this section are quite simple—and quite conservative. It is not necessary that the domain be well-ordered everywhere—just on those parts that are used to define the range.

10.3.2 Very dependent recursive types

Another avenue we can explore is defining general very-dependent recursive types. By very-dependent recursive types, we mean the normal parameterized recursive type $\mu\,(i, X.T[i, X]; v)$, where the type construction T is allowed to depend on X belonging to the recursive type definition. In the characterizations that we explore, we do no enforce a monotonicity condition on the type construction T, with the conse-

quence that out characterizations are not based on a fixed–point.

The first characterization is one that requires no extension to the type theory. Instead of defining a new primitive type constructor, we assign an index of unrolling to each element of the type, and we use the type

$$i{:}\mathbb{N} \times \mathit{unroll}\,(x, X.T[i, X]; v)^i$$

where we define

$$\begin{array}{rcl} \mathit{unroll}\,(x, X.T[i, X]; v)^0 & \equiv & \mathit{Void} \\ \mathit{unroll}\,(x, X.T[i, X]; v)^i & \equiv & T[v, \lambda v.\mathit{unroll}\,(x, X.T[i, X]; v)]^{i-1} \end{array}$$

Essentially, the inhabitants are required to carry along the index of the unrolling that they belong to, and if the indexing is to be relatively transparent, the constructors must modify the indexes. For instance, the type of lists of monotonically increasing sequences of integers would be the type:

$$\begin{array}{l} \mu\,(T.\mathit{Unit} + l{:}T \times \mathit{case}\,l\ \mathit{of}\ \mathit{inl}\cdot \Rightarrow \mathbb{Z} \mid \langle _, i\rangle \Rightarrow \{i+1\ldots\}) \equiv \\ \quad i{:}\mathbb{N} \times \mathit{unroll}\,(T.\mathit{Unit} + l{:}T \times \mathit{case}\,l\ \mathit{of}\ \mathit{inl}\cdot \Rightarrow \mathbb{Z} \mid \langle _, j\rangle \Rightarrow \{j+1\ldots\})^i \end{array}$$

The [] element is $\mathit{inl}\cdot$ and the *cons* operation is

$$h :: t \quad = \quad \mathit{case}\,t\ \mathit{of}\ \langle i, l\rangle \Rightarrow \langle i+1, \mathit{inr}\,\langle l, h\rangle\rangle$$

Induction of lists becomes induction on their lengths.

While this coding is sufficient for most constructions, it is intrinsically unsatisfying. The indices of unrolling are used only so that we can give the construction a type. In fact, there is enough information in the construction that we could compute the index if necessary. The requirement of indices conflict with a philosophy we normally adopt: programs should not be required to compute type information. The reason for this philosophy is the thought that the typing of an algorithm should not interfere with its efficiency; the computation of type information is inherently useless.

There are other pssiblities. For instance, we can equate the meaning of a recursive type with its infinite unrolling, or with the union of all its unrollings.

10.4 A Theory of Objects

For the next step, we will use the very-dependent function to give an interpretation of objects.

$$
\begin{array}{rcl@{\qquad}rcl}
z & \equiv & [l_i = \lambda x_i.T_i\ ^{i\in\{1\ldots n\}}] & & & \\
Z & \equiv & Obj(X.[l_i\nu_i\colon T_i[X]\ ^{i\in\{1\ldots n\}}]) & Z_i & \equiv & Obj(X.[l_j\nu_j\colon T_j[X]\ ^{j\in\{1\ldots i\}}]) \\
Z' & \equiv & Obj(X.[l_i\nu_i\colon T'_i[X]\ ^{i\in\{1\ldots n+m\}}]) & Z'_i & \equiv & Obj(X.[l_j\nu_j\colon T'_j[X]\ ^{j\in\{1\ldots i\}}])
\end{array}
$$

$$
\frac{\Gamma, o\colon Z_{i-1}, X_1, X_2\colon Type, X_1 \subseteq X_2 \vdash T_i[o/\bar{l}, X_1/X] \subseteq T_i[o/\bar{l}, X_2/X]}{\Gamma \vdash Z \in Type} \quad \textbf{Obj wf}
$$

$$
\frac{\Gamma \vdash Z \in Type \quad \Gamma, T \preceq Z_{i-1}, o\colon T \vdash t_i[o/x_i] \in T_i[o/\bar{l}, T/X]}{\Gamma \vdash z \in Z} \ \textbf{Obj mem}
$$

$$
\frac{\Gamma \vdash o \in Z}{\Gamma \vdash o.l_i \in T_i[\bar{l}, Z/X]} \ \textbf{Obj sel}
$$

$$
\frac{\Gamma \vdash Z \in Type \quad \Gamma, T \preceq Z_{i-1}, o\colon T \vdash t[o/x] \in T_i[o/\bar{l}, T/X]}{\Gamma \vdash o\,[l_i \leftarrow \lambda x.t] \in Z} \ \textbf{Obj ovr}
$$

$$
\frac{\begin{array}{l}\Gamma \vdash Z \in Type \\ \Gamma \vdash Z' \in Type \\ \Gamma, o\colon Z'_{i-1} \vdash T'_i[o/\bar{l}, Z'_{i-1}/X]^{\nu'_i} \lhd T_i[o/\bar{l}, Z_{i-1}/X]^{\nu_i} \text{ for } i \in \{1\ldots n\}\end{array}}{\Gamma \vdash Z' \preceq Z} \quad \textbf{Obj sub}
$$

$$
\frac{\Gamma \vdash S = T \in Type}{\Gamma \vdash S^{\circ} \lhd T^{\circ}} \ \textbf{Sub Inv} \qquad \frac{\Gamma \vdash S \preceq T \text{ for } \nu \in \{^{\circ}, ^{+}\}}{\Gamma \vdash S^{\nu} \lhd T^{+}} \ \textbf{Sub Cov}
$$

Figure 10.3: Rules for dependent calculus

10.4.1 Formal components of an "object–system"

By the term "object–system," we mean any system that provides the following components:

1. Abstract data types, specified by signatures, consisting of "methods" that implement components of the abstract data type.

2. Subtyping, with automatic inheritance of the methods of an abstract data type.

3. Optionally, method override, where methods may create copies of an object with new values for some of the methods.

There are other features that are common to many object–oriented implementations that we do not attempt to explain. For instance, many typed object-systems also provide operator overloading (also called "ad-hoc polymorphism"). In some systems method invocation is treated as message passing. While these features mesh well with an object–oriented style, they do not occur in all implementations, and they don't seem to be essential, as opposed to the modularity and inheritance features, which are common to all object systems. While we do not provide these extra features, our treatment will not preclude them.

Method override is one of the more controversial components of the object system, because of its semi–imperative nature. Method override is useful in a system where the methods of an object are allowed to refer to one another in their implementations. When a method is overridden, the other methods of the object are updated automatically to refer to the new value. Despite its imperitive nature, the functional coding is quite reasonable. In the first–order object system of this section we will include method override, and method interdependence, as practical features of the object system.

Method interdependence is really a practical feature of the object system, since any implementation can be unfolded to remove the dependencies. In later sections of this paper, we will deal with domains where method override does not make sense, and we will not include it as a feature of the object system.

To handle method interdependence, we include a self–application within a method invocation. In the literature [AC94b, MHF93], and in many implementations, objects are treated as records, and method invocation is then a composition of field selection and self–application. This gives us an untyped object calculus of the following form:

Let o be the object $\{l_i = \lambda x_i.b_i\}$ (l_i distinct for $i \in I$)

Method selection: $o.l_j = o[l_j](o) = b_j\left[o/x_j\right]$

Method override:
$$\begin{aligned} o.l_j \leftarrow \lambda z.m &= o\left[l_j \leftarrow \lambda z.m\right] \\ &= \{l_j = \lambda z.m, l_i = \lambda x_i.b_i\} \quad (i \neq j) \end{aligned}$$

Let's consider an example that seems common to much of the object–oriented literature: a two dimensional point in $\mathbb{Z} \times \mathbb{Z}$. The point $p = (-1, 7)$ would be the following object:

$$p = \{x = \lambda x. -1; y = \lambda x.7\}$$

The methods x and y are constant methods. Retrieving the x coordinate of the point p is a field selection, plus a self–application:

$$p.x = p[l_x](p) = (\lambda x. -1)\,(p) = -1$$

Now consider the class of movable points *MovablePoint*, which is a *Point* together with a functional method *Move* that produces a new point relative to the original point. The *Move* function computes the new position from existing x and y methods, and overiddes them with new values. The point $p = (-1, 7)$ can be implemented as:

$$\begin{array}{l} p = \{ \\ \quad x \quad = \quad \lambda s. -1 \\ \quad y \quad = \quad \lambda s.7 \\ \quad \mathit{Move} \quad = \quad \lambda s, i, j.s\,[x \leftarrow s.x + i]\,[y \leftarrow s.y + j] \\ \} \end{array}$$

Next, we wish to give the signature type. The obvious signatures are the following:

$$\begin{array}{l} \mathit{Point} = \{ \\ \quad x\colon \mathit{Point} \to \mathbb{Z} \\ \quad y\colon \mathit{Point} \to \mathbb{Z} \\ \} \end{array}$$

$$\begin{array}{l} \mathit{MovablePoint} = \{ \\ \quad x\colon \mathit{MovablePoint} \to \mathbb{Z} \\ \quad y\colon \mathit{MovablePoint} \to \mathbb{Z} \\ \quad \mathit{Move}\colon \mathit{MovablePoint} \to \mathbb{Z} \to \mathbb{Z} \to \mathit{MovablePoint} \\ \} \end{array}$$

Clearly, we are going to have a problem with these signatures. Making the types recursive will be a feat because they are anti-monotonic. One of the possible solutions is to hide the negative occurrence of the

signature type definition by providing a primitive signature type constructor. Yet, there are still problems deciding when a signature is well–formed, because of the seemingly necessary condition that there be an object belonging to the signature in order to prove well–formedness of the the signature type (due to the self application).

Another technique that will prove very useful is to use a bounded fixed-point $\mu\,(T' \subseteq T.B[T'])$, where

$$\mu\,(T' \subseteq T.B[T']) \equiv \mu\left(T.\bigcap_{T' \subseteq T} B[T']\right).$$

Such fixed points are trivially monotonic. Furthermore, they are useful for expressing polymorphism of the defined types. We will be using technique in our development.

From here we develop a type–theoretical account of the object system. Using the expressive power of the type theory, and the dependent record extension, we can develop a type for signatures in a normal type–theoretic, well–founded manner. Before we proceed with this, we will give a brief discussion the record calculus that we build on top of the very-dependent function type.

10.4.2 Polymorphism and record extension

We discuss a slight modification to the object–system before we proceed, to deal with polymorphism and record extensions. Generally, methods will be polymorphic over all subtypes of the signature that the method is specified in. We wish to capture this polymorphism in the type system. Consider the type *MovablePoint* that we described in section 10.4.1:

$$\begin{array}{l}
\mathit{MovablePoint} = \{ \\
\quad x\colon \mathit{MovablePoint} \to \mathbb{Z} \\
\quad y\colon \mathit{MovablePoint} \to \mathbb{Z} \\
\quad \mathit{Move}\colon \mathit{MovablePoint} \to \mathbb{Z} \to \mathbb{Z} \to \mathit{MovablePoint} \\
\}
\end{array}$$

Using the incremental construction of the previous section, this signature would be composed of three sub–signatures:

$\mathit{MovablePoint}_x$ `is` {}	$\mathit{MovablePoint}_y$ `is` {}
method $x\colon \mathit{MovablePoint}_x \to \mathbb{Z}$	**method** $y\colon \mathit{MovablePoint}_y \to \mathbb{Z}$
$\mathit{MovablePoint}$ `is` $\{\mathit{MovablePoint}_x, \mathit{MovablePoint}_y\}$	
method $\mathit{Move}\colon \mathit{MovablePoint} \to \mathbb{Z} \to \mathbb{Z} \to \mathit{MovablePoint}$	

$$\begin{array}{rcl} \text{Let} \quad z & \equiv & [l_i = \lambda x_i.t_i \ ^{i\in\{1\ldots n\}}] \\ Z & \equiv & Obj(X.[l_i \nu_i\colon T_i[X] \ ^{i\in\{1\ldots n\}}]) \\ Z' & \equiv & Obj(X.[l_i \nu_i\colon T'_i[X] \ ^{i\in\{1\ldots n+m\}}]) \end{array}$$

$$\frac{\Gamma, X_1, X_2\colon Type, X_1 \subseteq X_2 \vdash T_i[X_1/X] \subseteq T_i[X_2/X] \quad \text{for } i \in \{1\ldots n\}}{\Gamma \vdash Z \in Type} \ \textbf{Obj wf}$$

$$\frac{\Gamma \vdash Z \in Type \quad \Gamma, T \preceq [l_i \nu_i\colon T_i[T/X] \ ^{i\in\{1\ldots n\}}], o\colon T \vdash t_i[o/x_i] \in T_i[T/X]}{\Gamma \vdash z \in Z \quad \textbf{Obj mem}}$$

$$\frac{\Gamma \vdash o \in Z}{\Gamma \vdash o.l_i \in T_i[Z/X]} \ \textbf{Obj sel}$$

$$\frac{\Gamma \vdash Z \in Type \quad \Gamma, T \preceq [l_i \nu_i\colon T_i[T/X] \ ^{i\in\{1\ldots n\}}], o\colon T \vdash t[o/x] \in T_i[T/X]}{\Gamma \vdash o\,[l_i \leftarrow \lambda x.t] \in Z \quad \textbf{Obj ovr}}$$

$$\frac{\Gamma \vdash Z \in Type \quad \Gamma \vdash Z' \in Type \quad \Gamma \vdash T'_i[Z']^{\nu'_i} \lhd T_i[Z]^{\nu_i} \text{ for } i \in \{1\ldots n\}}{\Gamma \vdash Z' \preceq \quad \textbf{Obj sub}}$$

$$\frac{\Gamma \vdash S = T \in Type}{\Gamma \vdash S^{\circ} \lhd T^{\circ}} \ \textbf{Sub Inv} \qquad \frac{\Gamma \vdash S \preceq T \text{ for } \nu \in \{^{\circ}, ^{+}\}}{\Gamma \vdash S^{\nu} \lhd T^{+}} \ \textbf{Sub Cov}$$

Figure 10.4: Rules for nondependent calculus

In each of the sub–signatures, the method type refers to its enclosing signature. Since the type definition is anti–monotonic in the signature, we would not expect a normal fixed–point construction. The types in this example do not in fact need to use a self–reference. The arguments to the methods x and y can have any type, and the *Move* method is polymorphic over all records having integer methods x and y. In general, we will allow a method to override itself. If a method refers to itself, it should be polymorphic over subtypes of its own signature, or if not, it should be polymorphic over subtypes of the signatures of its parents.

Of course, even a method that overrides itself need not refer to itself using the "self" argument. If it wishes to refer to itself, it can use a standard fix-point, or Y–combinator construction. Thus the "self" argument to a method can always have the type that is the join of its parent signatures, and we never need consider an anti–monotonic recursive construction. However, monotonic definitions may appear more useful. For instance, consider non–empty lists of *Point*s, which might have the following signature:

$Point_x$ `is` $\{\}$ **method** x: $Top \to \mathbb{Z}$	$Point_y$ `is` $\{\}$ **method** y: $Top \to \mathbb{Z}$
Point `is` $\{Point_x, Point_y\}$ **method** *void*: $Top \to Unit$	
$PointList_{tail}$ `is` $\{Point\}$ **method** *Tail*: $Top \to PointList_{tail}$	
PointList `is` $\{PointList_{tail}\}$ **method** *Cons*: $\bigcap_{T \subseteq Point} \bigcap_{T' \subseteq PointList_{tail}} T \to T' \to T'$	

In this signature, the *Tail* method is either $\cdot$ or it is a subtype of *Point*. The *Cons* method adds a point to a list. Even in this case, it is possible to weaken the type of *Tail* to $Top \to Point$ because the *Cons* method has a more constrained type. In practice, we may find that a recursive formulation is never needed.

On the other hand, let's consider a recursive object system. We will maintain the structure of the objects of the last section, and modify the types to account for the recursion. The subtypes that we need are just the *record extensions*. A record extension for a record r is a record that has more fields than r, but with the same type of labels. A general subtype of r allows the label types to grow. An extension of a signature is a signature with more methods, but the same type of method labels. For each signature extension there is a corresponding signature *description* extension. Our recursive sobject system will be based on extended signature descriptions.

The method types depend on polymorphism over record extensions, so we must allow methods to use extended descriptions to compute their type. The new *MethodDescription* is:

$$\begin{array}{l}\texttt{type}\ \mathit{MethodDescription}(f, T) == \{\\ \quad \mathit{parents}: \mathit{Atomlist}\\ \quad \mathit{method}: \mathit{Signature}\,(f, \mathit{parents})) \to T \to \mathit{Type}\\ \}\end{array}$$

Where f is a *SignatureDescription* of the join of the parents, as before, and T is the type of all record extensions of the signature description of this method.

The new *SignatureDescription* requires a bounded, recursive type construction.

$$\begin{array}{c}\texttt{type}\ \mathit{SignatureDescription} ==\\ \mu\,(T \subseteq S.\,\{f \mid \mathit{name}: \mathit{Atom} \to \mathit{MethodDescription}(f, T)\})\end{array}$$

The *Signature* function changes slightly, as it must provide a signature description to methods, in order to compute their type.

This new form of recursive type is slightly more dubious than the last. We haven't attempted a formal proof, and although the intuition here seems correct, it not not clear that that we can produce a well–formed type. This is an interesting area for further research. In the next section we discuss constructors for signature and method descriptions.

10.4.3 Propositional meaning of signatures

As we will see in the next section of this paper, object signatures have an interesting Curry–Howard interpretation as propostions. The method types of the signatures are *axioms* of the signature, where the object is an implementation of those axioms. The implementation of an object may further extend the signature with theorems that are derivable fro its axioms. Within the object, the axioms are general properties that hold, and the theorems can be trated as axioms that have been proven. In other words, a theorem is a method that has a default implementation provided. As we would expect, further refinements of may override the implementation provided by the theorem with a different, possibly more efficient value.

This suggests a model where the construction of signatures proceeds in the same way that a theory is constructed. The axioms of the theory are stated, and then theorems are proved based on the axioms. The signature of the theory is record consisting of its axioms, and an object of the theory is a record of proofs of the axioms. Thus we have two

models for the object system, one is the standard algebraic model of abstract data types (with re–use), and the other is a logical model based upon *theories* or *book* of the logic. These are not competing views; they are complimentary, and within the standard type theory the views are isomorphic, and we can compute one from the other.

10.4.4 A record calculus

The record calculus we will use to construct objects has a simple encoding on top of the very–dependent function type. A record has a following features: it has a collection of labels, which are usually just names, although arbitrary types are permissable. The record allows access to its fields by name, and it provides an operation to replace a field by new value. This has a natrual encoding using a function type as follows:

Let the record type be **Record** $(A \to B)$ where the domain of labels is A and the domain of values is B. Let $r \in$ **Record** $(A \to B)$	
Record coding:	**Record** $(A \to B) \equiv A \to B$
Field selection:	$r_a \equiv r(a)$ (where $a \in A$)
Field replacement:	$r\,[a \leftarrow b] \equiv \lambda i.$ **if** $i = a$ **then** b **else** $r(i)$ (where $a \in A$ and $b \in B$)

For dependent records, we have a slightly different encoding, because the domain of labels must be well–ordered and the value types depend on their label and the record itself. In this case, the type for dependent records **Record** $\{r \mid a{:}A \to B\}$ is the very–dependent type $\{r \mid a{:}A \to B\}$. The coding for field select and replacement does not change.

10.4.5 Object system

We can develop a standard collection of constructors for signatures and object descriptions that give a more elegant notation to their descriptions. We would like to have the following features:

1. Each method type is defined, it is defined in a context where subtypes of the ancestors are represented by their names. The *Self* type is understood.

2. A signature is automatically constructed at the time a method is defined.

We can perform these operations formally within the system because signature descriptions are formal objects. When a method is defined, it is possible to walk the signature descriptions of the parents, join them, and extract the signature names for each of the ancestors. The new method can be defined in the scope where the ancestor names are bound to the intersection of their record extensions.

To define a new signature with a method, we define the followng operation.

NewSignature ⟨*name*⟩ **inherits** ⟨*parentlist*⟩ **method** ⟨*name*⟩ : ⟨*method-type*⟩

If the **inherits** field is omitted, then there are no ancestor signatures, and if the **method** field is omitted, it defaults to the identity function. Using this template, we can define the *MovablePoint* example as follows:

NewSignature $Point_x$ **method** x: $\mathbb{Z}$	**NewSignature** $Point_y$ **method** y: $\mathbb{Z}$
NewSignature *Point* **inherits** [$Point_x$; $Point_y$]	
NewSignature *MovablePoint* **inherits** [*Point*] **method** *move*: $Point \to \mathbb{Z} \to \mathbb{Z} \to Point$	

10.4.6 Types for an object system

At this point we are prepared to give an object system. We will build signatures recursively, one method at a time. The signature for a module will be an incremental construction of partial signatures for the module. Each signature has a collection of *parent* signatures that it inherits methods from, and an additional method type that may depend on the parent signatures as well as the signature being defined. We get a following template for a signature:

Parent function:	*parents*
ParameterType:	*Type*
Method:	*Type*

As we mentioned in section 10.2, we use the template to define a type of *method descriptions*, from which the signatures can be computed. We will collect the method descriptions into a single database, which we will implement as a record that specifies a complete signature.

Each method is assigned a unique label in the database record, and the database is ordered so that each template may refer to the descriptions of its parents. For the moment, the labels of the database record are inhabitiants of the type *Atom*, which consists of finite length strings of characters. Given the template for the signature, the database record of the parents f, and a list of parent names, we get the following type for a method description:

```
type MethodDescription(f) == {
  parents: Atom list
  method: Signature(f, parents)) → Type
}
```

The *Signature* function computes the signatue of the *parents* from the signature description f. This signature description record has the following form:

```
type SignatureDescription ==
  {f | name: Atom → MethodDescription(f)}
```

These two specifications are the key parts of the definition of formal objects. The *MethodDescription* specifies a particular method of an object, and *SignatureDescription* specifies an entire object. The *Signature* function, shown in Table 10.1, computes the signature from its description. Since we are implementing objects as dependent records, the signature is a dependent record type, which we can build by collecting the ancestors of the object.

Each method is added only once to the record. When a method is to be added (with name in the *name* argument), it is added only if it isn't in the *domain* list. If it hasn't already been added, its parents are added recursively, and then the *object* record is extended by a new field with key *name* and and type $method(object')$, where *method* is the "method" value in the signature description, and $object'$ is the object that has been extended to include the parent objects.

The *Join* function terminates because the signature description for each object is finite: assign each *name* in the *names* argument to *Join* with metric 0 if the signature description corresponding to *name* has no parents, or the maximum of the parent metrics plus one otherwise. Each description has a finite number of parents, so this maximum exists. To order the *names* arguments, let $names > names'$ if m and m' are the list of metrics for *names* and $names'$, sorted in decreasing order, and m is lexicographically larger than m'. In each call to *Join*, the head *name* in *names* is removed, and possibly replaced by the list of its parents. If *name* has metric n, then it is replaced with a finite number of names

Table 10.1: Signature function

```
let Signature f names =
    let Join object domain [] = ⟨object, domain⟩
    and Join object domain (name :: names) =
          if name ∈ domain then
            Join object domain names
          else
            let sig = f(name)
                names′ = names@sig.parents
                ⟨object′, domain′⟩ =
                Join object domain names′
            in
                (object′ [name ← sig.method (object′)], name :: names′)
    and ⟨object, domain⟩ = Join {} []names
    in
        object
end
```

The auxiliary function *Join object domain names* constructs the signature, with the following arguments:

object:	the partially constructed record,
domain:	a list of labels in the partially constructed record,
names:	the labels to be added to the record,
f:	the signature description record

with metric $n - 1$, respecting the order on the name list. Since no name has metric less than 0, the function must terminate after a finite number of recursive unrollings.

We must still prove that the *Signature* function, and the record types *MethodDescription* and *SignatureDescription* have well–formed types. This uses an inductive argument, similar to the argument for termination of *Join*. In the base case, if an object has no parents, then *Signature* computes to *Unit*, so the *MethodDescription* and *SignatureDescription* for the method are well–formed. Otherwise, assume that the descriptions of the parents of a method have well-formed types. Then *Signature* computes to the join of the record types for the parents, which is a well–formed type, and so the *SignatureDescriotpn* and *SignatureDescription* for that object are well–formed.

This argument has been partially carried out formally. It is a difficult formal proof because of the recursive nature of the description and the inductive argument that is required. The proof is fairly easy for a modification to this system, where multiple inheritance is allowed, but common ancestors are not joined. Although the proof of the modified system is fairly easy, multiple inheritance is a fundamental feature that we wish to support, *with* the joining of common ancestors. The extended proof is still partial.

These two types, *MethodDescription* and *SignatureDescription*, and the function *Signature* are the main tools that we will use to implement the object system. In the next section we show how to implement a standard view of an object system using these tools.

10.5 Type theory and “theories”

In the preceeding section, we saw how to describe objects in the type theory, and we noted their similarity to “theories” within the type theory. There is one difference that we explore in the following sections. Some *theories* assume that their axioms are not only true, but they are well–formed. The type theory itself is an instance of this case. The axioms are proved true and well–formed according to some meta–theory, and it often doesn’t even make sense to prove that an axiom is well–formed. There is no type for these theories. In this section, we show how the meta–type theory can be extended, without modifying the semantics, so that theories will correspond to meta–objects, and the meaning of an object corresponds to the meaning of the theory, according to the Curry–Howard correspondence.

10.5.1 Type theory foundations

Let's consider the fundamental components of a *constructive* type theory. The type theory usually assumes some sort of functional computation system (most commonly the λ-calculus). The type theory does not usually distinguish between β-equivalent terms in the computation system (although there may be special-purpose systems with finer granularity that deal with computational complexity classes of terms). The computational terms exist *before* the types exist.

The types are added to the theory next, in an effort to classify the computational terms. The computational terms exist regardless of whether they have a type or not, but the type system imposes a meaning on the terms. For instance, some terms may abstract a computation over an argument (as a λ abstraction). The type system classifies some of these, and calls them *functions*.

In a constructive type theory, the type system may impose an additional requirement that the types denote predicates in a logic. This is the *Curry-Howard* isomorphism, and it describes a correspondence between the terms of a type and the proofs of a proposition.

Next, the type theory contains a binary equality, coarser then the computational equality, to describe what it means for two terms to be equal, or for two types to be equal.

Finally, the type theory contains rules, which are axioms that partially describe when term can be classified with a type, or when two terms or types are equal. A rules consists of *sequents*,

$$\overline{x}{:}\overline{H} \vdash \overline{T} \quad j{:}\,\mathtt{ext}\,(\overline{x})\,,$$

where $\overline{x}{:}\overline{H}$ is a dependent record (telescope) or types, $\overline{T}$ is a collection of types. The elements $x{:}H$ left of the turnstile are called *hypotheses*, and the types $\overline{T}$ right of the turnstile are called *conclusions*. Let I be the index set of the hypotheses, with well-founded order $<_I$, and let J be the index set of the conclusions, with well-founded order $<_J$. Then the *meaning* of the sequent is roughly as follows:

- *Assume* for each $x_i{:}H_i$ where $i \in I$, that H_i is a type, inhabited by a term denoted by x_i, under the same assumption for all $x_k{:}H_k$ where $k <_I i$.
- Then, one of T_j for $j \in J$ is inhabited by the term $\mathtt{ext}\,(\overline{x})$ (identically, T_j is true with proof corresponding to $\mathtt{ext}\,(\overline{x})$).

The sequent is a *subtype* relation, stating that if we have terms inhabiting the hypotheses, than there is an algorithm for generating a term inhabiting a conclusion. Since the algorithm decides which conclusion

is true, the rules are usually reduced to form involving a single conclusion.

The Nuprl logic refines the meaning of the sequent even more. In this case, the meaning of the sequent describes the equality of a type as follows:

- *Assume* for each $x_i{:}H_i$ where $i \in I$, that $H_i[\overline{x_k}]$ is true, where $\overline{x_k}$ is the vector of all x_k for $k < i$, and $H_i]\overline{x_k}] = H_i[\overline{x_k}']$ for $\overline{x_k} = \overline{x'_k}$ according the the types $\overline{H_k}$.
- Then, one of $T_j[\overline{x}]$ for $j \in J$ is inhabited by the term $\mathtt{ext}\,(\overline{x})$, and $T_j[\overline{x}] = T_j[\overline{x}']$ for equal $\overline{x} = \overline{x}'$ according to the assumptions $\overline{H}$.

Thus, the sequent refines the equality of the type T_j, and $\mathtt{ext}\,(\overline{x})$ is an algorithm for generating a term in T_j given terms $\overline{x} \in \overline{H}$.

A rule reflects the sequent onto itself; a rule has the form

$$\overline{x}{:}\overline{S} \vdash \overline{G} \quad j{:}\mathtt{ext}\,(\overline{x})\,,$$

where $\overline{x}{:}\overline{S}$ and $\overline{G}$ are vectors of sequents. The $\overline{x}{:}\overline{S}$ are called *subgoals*, and $\overline{G}$ are called *goals*, terminology taken from backward-chaining theorem provers. Just as for the sequent, there is no increase in generality in having more than one subgoal, so the goal is usually reduced to a single instance, and the rule has the form,

$$\overline{x}{:}\overline{S} \vdash G \quad \mathtt{ext}\,(\overline{x})\,.$$

The *meaning* of the rule is different from the sequent; the rule is a simple proof-theoretical fact. In the usual case, there are a finite number of subgoals, and each subgoal is distinct from the other subgoals (each subgoal is necessarily closed). The rule is a simple statement of the following form:

> If each subgoal is true, the the goal is true, and $\mathtt{ext}\,(\overline{x})$ inhabits its conclusion.

There is no need to ascribe a deeper meaning to the rule, because the deeper meaning of the sequent already provides the semantic foundation for the type theory.

The standard account of the type theory stops here. Any system implementing the type theory has the following organization:

1. there is a fundamental computation system,
2. there is a standard collection of rules that define the type theory,

3. everything else is either definition or a theorem, in a global arena provided by the type theory.

Yet, an alternative account seems natural, especially if we wish to work with more than one type theory:

1. there is a fundamental computation system,
2. for each type theory,
   (a) there is a set of rules defining the type theory,
   (b) everything else is a definition or theorem based on the rules.

In fact, each type theory is like a rule: it assumes a collection of rules, and then proves a collection of theorems based on those rules. We can account for this extended notion of a type theory by generalizing rules. As a new definition, we generalize a sequent recursively. The class of sequents is described by S by the following grammar:

$$\begin{array}{rcll} S & ::= & \mathit{Term} & \\ & | & \overline{x}{:}\overline{S} \vdash S & \texttt{ext}\,() \end{array}$$

The meaning of closed sequents $s \in S$ is as follows:

- If $s \in \mathit{Term}$, then s denotes a type that is inhabited.
- Otherwise s is non-trivial sequent. Let the index set of the hypotheses be I with well-founded order $<_I$, and let

  $$s \equiv \overline{x_i}{:}\overline{s_i} \vdash g.$$

  Assume the following, for each $x_i{:}s_i$:

  { if $s_i \in \overline{y}{:}\overline{S} \vdash G$, then s_i is true, and x_i is an untyped term standing for the `ext` term of the sequent.

  { if $s_i \in \mathit{Term}$, then s_i is a type inhabited by the term x_i. Furthermore, s_i is functional over $x_j{:}s_j$ for $j <_I i$. If s_j is not a term, this functionality is trivial, since x_j is a unique term. Otherwise, for $x_j = x'_j$ in s_j, functionality implies $s_i(x_j) = s_i(x'_j)$.

  Under these assumptions, g is a type inhabited by $\texttt{ext}\,(\overline{x})$. Furthermore, g is functional, so that for $\overline{x} = \overline{x'}$ inhabiting the hypotheses, $\texttt{ext}\,(\overline{x}) = \texttt{ext}\,(\overline{x'})$ in the type g.

The argument that this is a sensible semantics for the generalized sequent is left for further study. For now, we just accept this semantics as the definition of the type system.

The generalized sequent can be used as a basis for all statements within the type theory. In the following sections, we cover templates for sequents (which are the standard forms of assertions), and then we explore the objects of the type theory, and we show their representations a generalized sequents. The type of "objects" will also be a form of generalized sequent.

10.5.2 Sequent templates and proofs

Using generalized sequents as a basis for the type theory can easily get out of hand. For example, a natural rule of a type theory is probably the AXIOM rule, which can be described as follows:

$$\overline{a}\colon\overline{A}, b\colon B[a], \overline{c}\colon\overline{C[a,b]} \vdash B[a] \quad \mathtt{ext}\,(b)\,.$$

This form states that if the goal is identical to a hypothesis, the goal is proved by the term inhabiting the hypothesis. This seems like a natural rule, but if we are restricted to generalized sequents, we require an infinite number of sequents to define the rule, one for each instantiation of $\overline{A}$, B, and $\overline{C}$. Thus, the normal method of expressing rules is by using templates, like the one above. The variables $\overline{A}$, B, and $\overline{C}$ are slots that may be instantiated with arbitrary sequents, and the variables $\overline{a}$, b, and $\overline{c}$ are slots that can be instantiated with arbitrary variables. This method is used throughout the theory, and statements are usually made in the form of templates. It is understood that the templates are forms that are not part of the type theory, but their instantiations are.

If we examine this concept carefully, there seems to be some sort of impredicativity here. For instance, is the following sequent true?

$$\left[\overline{a}\colon\overline{A}, b\colon B[a] \vdash B[a] \quad \mathtt{ext}\,(b)\right], i\colon\mathbb{Z} \vdash \mathbb{Z} \quad \mathtt{ext}\,(i)$$

At first glance it seems that the sequent is valid, but upon further reflection, we see that the sequent can't be instantiated to a true sequent. The template assumption can't be "applied" to itself. Instead of the previous sequent, we should be proving the following:

$$\left[\overline{a}\colon\overline{A}, b\colon B[a] \vdash B[a]\,\mathtt{ext}\,(b) \vdash\right][i\colon\mathbb{Z} \vdash \mathbb{Z}\,\mathtt{ext}\,(i) \quad],$$

which has no problems with self-application.

Another form of description that is often used is a *tactic*, which

is a program that can be used to generate a sequent. In a backward-chaining system, the tactic is given the goal sequent as an argument, and it computes subgoals that are sufficient to prove the goal, along with a program that computes the term inhabiting the goal. Tactics are usually arbitrary programs, and can't be trusted to produce only true sequents. Thus, in addition to the subgoals, the tactic is required to compute a verification, which is essentially a proof tree where the steps of the proofs are all assumptions of the theorem.

But what is a proof? In order to define it, we need at least *one* proof rule. The choice we make is, as would be expected, a version of *modus-ponens*:

$$\frac{\overline{c}:\overline{C}, \overline{a_j}:\overline{A_j} \vdash A_i \quad \mathtt{ext}\,(a_i)^*}{\overline{c}:\overline{C}, \left[\overline{a}:\overline{A} \vdash B \quad \mathtt{ext}\,(b) \vdash\right] B \quad \mathtt{ext}\,(b)}$$

: Let I be the index set of $\overline{A}$ with order $<_I$. Then there are I sequents, one for each $i \in I$, with j ranging over all I where $j <_I i$.

Note that if $\overline{A}$ is empty, the proof terminates.

It is important to note the interaction between proofs and sequent templates. In general, the template stands for an infinite number of sequents. Thus it would seems that modus-ponens would generate an infinite number of subgoals. This would be a significant practical problem! But it is not necessary if an application of modus-ponens keeps sequents in template form, instantiating only those parts of the sequent that are refined by the goal. In essence, an application of modus-ponens requires a unification of the applied sequent to the goal, and the most general unifier is used to create the subgoals.

Of course, the use of sequent templates may still cause problems if an use of the sequent generates an infinite term corresponding to the proof. For instance, suppose we have a thinning rule:

$$\left[\overline{a}:\overline{A}, \overline{c}:\overline{C} \vdash D \quad \mathtt{ext}\,(d)\right] \vdash \left[\overline{a}:\overline{A}, b:B, \overline{c}:\overline{C} \vdash D \quad \mathtt{ext}\,(d)\right]$$

If we restrict B to match a single hypothesis, not a pattern, then an infinite number of applications of thinning would be required to thin a single template. If the proof term is modified in the operations, the resulting proof term may be infinite. This becomes another task of the type theory designer.

10.5.3 Objects of the type theory

In general, statements in a type theory are assertions under assumptions. The one assumption present in every statement of the type theory is the underlying computation system. We can specify the computation system in the type theory, but we must still assume that the computation system obeys its specification. For any particular type theory, the basic rules of the theory are presented next, under no additional assumptions. The rules themselves are assertions, and state their assumptions explicitly. To a user of the type theory, the rules and computation system form a basic assumption present in every statement that is made. The type theory is not usually listed in each theorem, but it forms an implicit assumption.

A theorem itself is a degenerate form of a sequent. If the theorem proves a proposition T, it is expressed as the sequent $\vdash T\,\mathtt{ext}\,(t)$ where t is the term corresponding to the proof of t. If the assumptions are listed explicitly, the sequent would have the form $\overline{R} \vdash T\,\mathtt{ext}\,(t)$ where $\overline{R}$ is the list of rules of the type theory. This sequent states that T is a type of the type theory, and it is inhabited by the term t.

As we noted before, *tactics* also represent a collection of sequents, and we could list the tactics as assumptions of the theorem. Doing so is not necessary, since the tactics provide no extra information, but we may wish to do so provides tactics that are *local* to a theorem. If tactics are written formally and verified, it is possible to treat a tactic as another, derived, instance of a sequent.

Every instance of an assertion in the logic is a generalized sequent, and each true generalized sequent is a *theory*. The axioms of the theory are listed to the left of the turnstile, and the theorems are listed to the right of the turnstile. In general, a theory does not have a type, but some subparts of the theory may have a type. If the parts of a sequent at typable, under a collection of typing assumptions, then there is a type equivalent to the sequent. This type will correspond to a signature, an instance of the axioms will be an object, and the sequent itself is a theorem proving its conclusions.

When a sequent is not typable, there is no corresponding signature. We can still view the sequent as the specification for an object, and some of the operations that make sense on objects, makes sense in the theory. For instance, method selection corresponds to the application of an axiom. Some operations do not make sense. for instance, method override in this setting can only replace non–typable assumptions with equal terms, which is no override at all. However, the general concept of the object system applies in this setting, and it is possible to compute the object view of a theory from its sequent, and vice–versa.

By noting this correspondence, we find that there is only one object in a type theory: a generalized sequent. There are several views: a sequent can stand for a rule of the logic, a theorem, a theory, or a signature. A tactic is a collection of sequents that can be computed. An extract of a sequent is an extract of a rule, a proof of a theorem, or a method implementation in an object. The view that a user takes of an object is a matter of preference; all views have the same underlying meaning.

10.6 Related-work

There has been widespread research into formal methods for object–oriented programming. Abadi and Cardelli [AC94b, AC94a] have defined several typed and untyped object calculi. They demonstrate how hard it is to find a semantics for an object–calculus, and they eventually use an axiomatic untyped semantics. They also model the method selection as a combination of field selection and self–application. In their work, they define a type *Self* that stands for the signature of the object being defined, and in order to type an object they must assume that the type is well–defined.

Mitchell, Honsell, and Fisher [MHF93] develop an object calculus based on the untyped lambda calculus, and develop a static type checking system to later develop equational reasoning about method bodies. Their type system allows methods to be specialized as they are inherited, by polymorphism over subtypes.

Much of the basic theory of object–oriented programming is based on record calculi, and the semantics of subtyping and record extension. Amadio and Cardelli [AC91] discuss subtyping in the presence of recursion. In general, recursive subtyping produces a type larger than simple “record extension.” Harper and Pierce [HP91] present a record calculus, and they apply some of their work to an ML–style module system [HL94]. Wand [Wan87, Wan89] consider type–inference in the setting of record extension and multiple inheritance, a property that we do not discuss in this paper, but is a useful practical tool. Breu [Bre91] applies the techiques of algebraic specification to objects and their signatures. Dybjer has produced results on dependent inductive types with a set theoretic semantics.

There is a wealth of information on formal object systems in Gunter and Mitchell [GM94]. Rémy, Wand, and Ohori and Buneman develop type inference in object systems. Cardelli, Mitchell, and Rémy develop record calculi. Kamin and Reddy give semantics models of inheritance in record calculi, and Cook, Hill and and Canning decouple inheritance

and subtyping, as in a language like Eiffel [Mey88].

There is relatively less work on an interpretation of *theories*. Basic and Constable [BC93] discuss logical frameworks. They present a meta–logic that is similar our presentation of theories in 10.5. In their framework, the language of the logic and its proofs are formal objects belonging to abstract data types in *some* meta–type–theory. In our account, we reduce this meta–type–theory to two components: the generalized sequent and modus–ponens. Pitts [Pit00] presents a view of theories as *categories*. In many ways, the categorical view is similar to an object approach, where signatures are viewed as categories.

Jackson [Jac95] has provided a rich formal environment in the Nuprl system [CAB$^+$86] and applied it to constructive abstract algebra. Jsckson's work provides the grounds for this proposal—due to the power of his system, it is possible to formalize large areas of mathematics. In doing so, we become aware that the domains we are formalizing are themselves objects that can be described formally. In a poor environment, where formalism is small, the issue would never arise. Our work is also an extension of Jackson's formalization of abstract algebra.

Chapter 11

A Predicative Interpretation of Objects

Predicative type theories are powerful tools for giving foundational interpretations of programming languages. Due to their explicit inductive construction, predicative type theories have multiple mathematical models that provide precise definitions of programming language features. However, not all features have predicative interpretations, and current interpretations of objects rely on impredicative type theories, such as Girard's System *F*, because of the difficulty in specifying a type for objects in the presence of self-application.

In this chapter we show that objects have a predicative interpretation. We show that predicativity is associated with method monotonicity, and that binary methods prevent the inductive type construction. Our interpretation differs from impredicative accounts by replacing the use of recursive types for objects with conditions for method polymorphism over the *self* type. We further give a propositional meaning to objects in the type theory, providing a calculus for formal verification. Our interpretation has been verified in the Nuprl predicative type theory.

11.1 Introduction

Object-oriented languages provide mechanisms for program modularity and reuse that simplify the task of programming enough that object-oriented programming is now common practice. Unfortunately, the theory of objects has lagged behind practice due, in part, to the difficulty of assigning a type to self-application in the presence of method override. Self application occurs during method selection when the entire object is passed to the method as an implicit "self" argument.

Recent type-theoretic accounts of objects [AC96, PT94, Bru94, GM94] have been encoded in impredicative type theory (mainly variants of Girard's System *F* [Gir72]), which allow self-application through an impredicative interpretation of the self parameter. In this paper, we show that the construction of objects has an interpretation in *predicative* type theory.

This is a stronger result because predicative theories make fewer assumptions and have more models. In particular, the Nuprl type theory that we use in this paper has multiple mathematical models, notably set theoretic [How96], PER models [All86, Men87], denotational models [Rez85], and others. In addition, our interpretation is extendable to foundational formal objects that encode *proofs* [HNSS98, Hic96], a critical step toward our goal of providing a modular and object-oriented logical framework [Hic97].

The object interpretation is more delicate but we get more information. For example, predicative theories have a strictly inductive construction [ML75, All87], so the account of objects is inductive. An interesting feature is that impredicative type recursion is replaced by method polymorphism. The explicit inductive construction is also valid in impredicative type theory.

Our interpretation has the following benefits.

- The predicative account provides a simple, precise, mathematical account of objects.
- Inheritance is rooted in method polymorphism; recursive types are not necessary.
- We rely on the formal foundations of the type theory to verify soundness of the interpretation. A formal proof of soundness is available online.

In our interpretation, object types have propositional meaning. We further extend the calculus with dependent method types to provide a basis for formal program analysis. We develop our account on the object calculus of Abadi et al. [AC96, ACV96], which gives a precise characterization of self application and method override. We show that the type system can be developed without the use of recursive types or weak sums, and without sacrificing subtyping properties. Method type dependencies are expressed using dependent records based on the very-dependent function type [Hic96], and method polymorphism over subtypes is expressed using intersection types.

$$
\begin{array}{lrcl}
\text{Values} & v & ::= & [l_i = \lambda x_i.t_i]^{\,i\in\{1\ldots n\}} \\
\text{Terms} & t & ::= & v \mid t.l \mid t\,[l \leftarrow \lambda x.t] \\
\text{Types} & T & ::= & \mathit{Obj}(X.[l_i\colon T_i[X]^{\,i\in\{1\ldots n\}}])
\end{array}
$$

Figure 11.1: Syntax of the object calculus

11.2 Review

The formalism we use for objects in this paper is a functional object calculus similar to the calculus of Abadi, Cardelli, and Viswanathan [ACV96], in which the only primitives are objects, method invocation, and method update. While these object calculi are simple and precise, they allow representations of class-based and object-based concepts, as shown by Abadi and Cardelli in the "Theory of Objects" [AC96]. One of the significant features of these calculi is that they are all based on *self-application*: methods receive the object as an argument when they are invoked.

Since we are giving a semantic characterization of programs, we use an untyped operational calculus that allows programs to be interpreted according to their context. The syntax of the source language is given in Figure 11.1. In this language, objects $o = [l_i = \lambda x_i.t_i]^{\,i\in\{1\ldots n\}}$ are finite collections of *methods* labelled with names $l_1, l_2, \ldots, l_n$ drawn from an infinite collection of names. The method bodies $\lambda x_i.t_i$ are functions, where the variable x_i is a binding occurrence representing the "self" parameter for the method named l_i (Abada et. al. use the notation $\varsigma x_i.t_i$ to emphasize the difference between method bodies and standard lambda abstraction).

There are two operations on objects. *Method selection*, written $o.l_j$, substitutes the object o for x_j in body t_j. *Method override*, written $o\left[l_j \leftarrow \lambda x.t\right]$, creates a new object from o, replacing the method $\lambda x_j.t_j$ with the new method $\lambda x.t$. This operational semantics is summarized in the following table:

For an object $o = [l_i = \lambda x_i.t_i]^{\,i\in\{1\ldots n\}}$:

$$
\begin{array}{rcl}
o.l_j & \rightarrow & t_j[o/x_j] \\
o\left[l_j \leftarrow \lambda x.t\right] & \rightarrow & [l_j = \lambda x.t, l_i = \lambda x_i.t_i\,{}^{\,i\in\{1\ldots n\}-\{j\}}]
\end{array}
$$

The types $\mathit{Obj}(X.[l_i\colon T_i[X]^{\,i\in\{1\ldots n\}}])$ are used to describe objects, where X is a type quantifier representing the type of the object itself,

and T_i is the type of the method named l_i relative to the object type X.

Our task is to provide an encoding of the three primitives and their types in the type theory. The type theory we are using is a small, but very expressive fragment of the Nuprl type theory including subtypes, intersections, dependent function types, as well as the terms of the untyped lambda calculus, numbers, and strings.

11.2.1 Record encoding

We interpret objects as records, where the record $\{l_1 = t_1, \ldots, l_n = t_n\}$ has fields $l_1, \ldots, l_n$ with values $t_1, \ldots, t_n$, respectively. Records provide selection (without self-application) written $r \cdot l$ for a record r on field l, and functional update, written $r \cdot l := x$, which creates a new record from r with the value of field l replaced by t. The record calculus is described in the following table:

For a record $r = \{l_i = t_i\}^{\,i \in \{1 \ldots n\}}$:

$$\begin{array}{rcl} r \cdot l_j & \rightarrow & t_j \\ r \cdot l_j := t & \rightarrow & \{l_j = t, l_i = t_i\}^{\,i \in \{1 \ldots n\} - \{j\}} \end{array}$$

The encoding of the object primitives in the record calculus is as follows:

$$\begin{array}{rcl} [l_i = \lambda x_i.t_i^{\,i \in \{1 \ldots n\}}] & \equiv & \{l_i = \lambda x_i.t_i^{\,i \in \{1 \ldots n\}}\} \\ o.l_j & \equiv & (o \cdot l_j)(o) \\ o\left[l_j \leftarrow \lambda x.t\right] & \equiv & o \cdot l_j := \lambda x.t \end{array}$$

We further encode records as functions on labels. For simplicity, let the collection of all labels have the type *Atom*, and assume records have as default value the type *Top* on uninteresting elements. The type *Top* is the equivalence class containing all terms (including the term *Top*).

$$\begin{array}{rcl} \{l_i = t_i^{\,i \in \{1 \ldots n\}}\} & \equiv & \lambda l.\,\textbf{case}\, l\, \textbf{of}\, l_1 : x_1 | \cdots | l_n : x_n | _ : Top \\ r \cdot l_j & \equiv & r(l_j) \\ r \cdot l_j := x & \equiv & \lambda l.\,\textbf{if}\, l = l_j\, \textbf{then}\, x\, \textbf{else}\, r \cdot l \end{array}$$

11.3 Object Interpretation

Using the encoding of objects as functions, we would expect that the object types $Obj(X.[l_i: T_i\ ^{i\in\{1\ldots n\}}])$ obey the equation

$$X = \{l_i: X \to T_i^{\,i\in\{1\ldots n\}}\}.$$

However, this type can't be expressed using general predicative recursive types because of the contravariant occurrence of X in the equation. For the moment, we restrict attention in this section to objects that have a well-founded call graph on the methods (objects with mutually recursive methods are considered in Section 11.3.3). To illustrate the properties of non-recursive objects, we fall back on the traditional example—the one-dimensional point, which has a field x that is an integer, and a method to produce a point shifted to the right by 1 unit.

$$\begin{array}{rcl} P & \equiv & Obj(X.[x: \mathbb{Z}; move: X]) \\ p_0 & \equiv & [x = \lambda o.0; move = \lambda o.(o\,[x \leftarrow o.x + 1])] \end{array}$$

Although the coding

$$[\![P]\!] = \mu(X.\{x: X \to \mathbb{Z}; move: X \to X\})$$

is not well-formed, the x method does not use its argument, and the *move* method refers only to x. A more specific typing is:

$$[\![P]\!] = \{x: \{\} \to \mathbb{Z}; move: \{x: \{\} \to \mathbb{Z}\} \to \{x: \{\} \to \mathbb{Z}\}\}.$$

This type is not recursive, but it contains objects whose move method "forgets" about all methods but x. In order to get the desired behavior, the typing for *move* must specify that it does not affect any other method but x. In other words, it should be polymorphic over all object types that are extensions of $\{x: \{\} \to \mathbb{Z}\}$.

In the untyped calculus, polymorphism is expressed with the intersection type. The typing

$$move \in \bigcap\nolimits_k T \subseteq \{x: \{\} \to \mathbb{Z}\}.T \to T$$

specifies that the *move* method has the typing $T \to T$ for any T that is a subtype of $\{x: \{\} \to \mathbb{Z}\}$ in the predicative type universe $\mathbb{U}_k$. Unfortunately, this typing also fails to produce the expected behavior because it contains only the identity functions [RT88]. Essentially, we don't want polymorphism over *subtypes* of $\{x: \{\} \to \mathbb{Z}\}$, but over record *extensions* of $\{x: \{\} \to \mathbb{Z}\}$. Since the *move* method modifies the value of the method

x, the type for x must be large enough to contain the modified value.

11.3.1 Constrained polymorphism

In order to address the subtyping problem, we must specify which methods are updatable. We adopt the method of *variance annotations* [ACV96], where method labels are annotated with the symbols ($^+$, $^-$, $^\circ$), which are intended to provide the extra information. Object types are covariant in the types of their $^+$ methods, contravariant in the types of their $^-$ methods, and invariant in the type of their $^\circ$ methods (we do not consider the annotation $^-$).

Given the type annotations, we design a new constrained subtyping relation $\preceq$ to respect the type annotations. The subtyping relation $\subseteq$ retains its original semantics, and we have $A \preceq B \Rightarrow A \subseteq B$. Given an object type

$$T = Obj(X.[l_i \nu_i : T_i[X]^{i \in \{1 \ldots n\}}])$$

for $\nu_i \in \{^+, ^\circ\}$, let $\mathbf{L}_T^{upd}$ be the labels of the methods in T that are invariant. The set $\mathbf{L}_T^{upd}$ contains only the labels mentioned explicitly in T, and it is defined as follows:

$$\mathbf{L}_T^{upd} \equiv \{l_i \in Atom \mid \nu_i = {}^\circ\}.$$

Let the notation $T|_{\mathbf{L}}$ be the restriction of the object type T to the methods named in the set $\mathbf{L}$. We define the relation $\preceq$ as follows:

$$T' \preceq T \quad \textit{iff} \quad T' \subseteq T \wedge T'|_{\mathbf{L}_T^{upd}} \supseteq T|_{\mathbf{L}_T^{upd}}.$$

Essentially, we consider the types T' that are subtypes of T *and also* supertypes of T on the invariant methods (so T' and T are extensionally equivalent on the invariant methods). The intuition here is that the relevant subtypes must not further constrain the types of the invariant elements.

To make this definition more explicit, in the type theory we require that the interpretations of the types T' and T be dependent function types of the form $l : Atom \to R_{T'}(l)$ and $l : Atom \to R_T(l)$, for some types $R'_T(l)$ and $R_T(l)$. We restrict the subtyping condition to the type func-

tions $R_{T'}$ and R_T to get the following definition:

$$\begin{array}{rcl} T' \preceq T & \textit{iff} & \exists R_T, R'_T\colon \mathit{Atom} \to \mathit{Type}. \\ & & \quad [\![T]\!] = (l\colon \mathit{Atom} \to R_T(l)) \\ & & \wedge\ [\![T']\!] = (l\colon \mathit{Atom} \to R_{T'}(l)) \\ & & \wedge\ \forall l\colon \mathit{Atom}. R_{T'}(l) \subseteq R_T(l) \\ & & \wedge\ \forall l\colon \mathbf{L}_T^{upd}. R_{T'}(l) \supseteq R_T(l) \end{array}$$

Note that this definition specifies that the methods of T' that do not appear in T are covariant relative to the object type T (there are *no* labels in $\mathbf{L}_T^{upd}$ that do not appear explicitly in the definition of T).

In the remainder of this chapter, we will also use the notation $S' \preceq_T S$ to be the subobject relation on types relative to an object definition T. That is,

$$\begin{array}{rcl} S' \preceq_T S & \textit{iff} & \exists R_S, R'_S\colon \mathit{Atom} \to \mathit{Type}. \\ & & \quad S = (l\colon \mathit{Atom} \to R_S(l)) \\ & & \wedge\ S' = (l\colon \mathit{Atom} \to R_{S'}(l)) \\ & & \wedge\ \forall l\colon \mathit{Atom}. R_{S'}(l) \subseteq R_S(l) \\ & & \wedge\ \forall l\colon \mathbf{L}_T^{upd}. R_{S'}(l) \supseteq R_S(l) \end{array}$$

11.3.2 Non-recursive interpretation

Given this definition of restricted subtyping, the translation of object types for non-recursive objects can be given. Since the method call graph is acyclic, we arrange the methods in the object so that each method refers only to the previous methods, i.e. the method named l_i refers only to methods $l_1, \ldots, l_{i-1}$.[1] We code the object type inductively for $T = \mathit{Obj}(X.[l_i \nu_i\colon T_i[X]^{i \in \{1 \ldots n\}}])$. Let T_i be the object type $\mathit{Obj}(X.[l_j \nu_j\colon T_j[X]^{j \in \{1 \ldots i-1\}}])$. The encoding for T is specified as follows:

$$\begin{array}{rcl} [\![T_0]\!] & = & \{\} \\ [\![T_{i+1}]\!] & = & [\![T_i]\!] \cap \left\{ l_{i+1}\colon \bigcap\nolimits_k T \preceq_{T_i} [\![T_i]\!].T \to T_{i+1}[T/X] \right\}, \end{array}$$

where $[\![T]\!] \equiv [\![T_n]\!]$. Note that record intersection for records with distinct nontrivial fields is the same as record concatenation, and that $[\![T_{i+1}]\!] \preceq_{T_i} [\![T_i]\!]$ follows from the properties of record extension.

[1] This ordering is done purely for exposition—it does not suppose that there is a single well-ordering ordering on the labels for all objects. Objects are considered equivalent if methods are reordered.

The issue of predicativity shows itself in the behavior of method selection. The rule we would like is the following:

$$\frac{\Gamma \vdash o \in T}{\Gamma \vdash o.l_i \in T_i[T/X]} \textbf{ Obj sel}$$

For this rule to be valid, the method being selected must preserve the type T of the self parameter. Given the object $o \in T$, the full type of method l_i is

$$[\![o]\!](l_i) \in \bigcap\nolimits_k T' \preceq_{T_i} [\![T_{i-1}]\!].T' \to T_i[T'/X].$$

By instantiating the quantifier with $[\![T]\!]$ itself, we would have

$$[\![o]\!](l_i) \in [\![T]\!] \to T_i[[\![T]\!]/X],$$

and $[\![o]\!](l_i)([\![o]\!]) \in T_i[[\![T]\!]/X]$, which is the desired result.

Unfortunately, even though $[\![T]\!] \preceq_{T_{i-1}} [\![T_{i-1}]\!]$, it is not possible to instantiate the quantification $\bigcap_k T' \preceq_{T_i} [\![T_{i-1}]\!].\cdots$ directly, because of the occurrence of the type universe $\mathbb{U}_k$. This occurrence forces the intersection to belong to $\mathbb{U}_{k+1}$, which in turn forces the object type $[\![T]\!]$ into $\mathbb{U}_{k+1}$, prohibiting $[\![T]\!]$ from being used to instantiate the intersection. This is the crucial impredicative occurrence of self application. Before addressing this problem, we generalize the calculus to objects with mutually recursive methods.

11.3.3 Recursive Objects

Objects with recursive methods require additional interpretation because of the need to specify the recursion in the method types. To illustrate the issue, consider the following example of a point that contains a position and also a method to move the point to the factorial of its position. The factorial point has the following type:

$$P_F \equiv Obj(X.[x^\circ\colon \mathbb{N}; fact^+\colon X]).$$

An example of such a point is the following object:

$$p = \left[\begin{array}{l} x \quad = \lambda o.5; \\ fact = \lambda o.\textbf{if } o.x = 0 \textbf{ then} \\ \qquad\qquad o\,[x \leftarrow 1] \\ \qquad\quad \textbf{else} \\ \qquad\qquad o\,[x \leftarrow (o\,[x \leftarrow o.x - 1]).fact.x * o.x] \end{array}\right]$$

Let $z \equiv [l_i = \lambda x_i.t_i \ ^{i\in\{1\ldots n\}}]$
$Z \equiv Obj(X.[l_i \nu_i\colon T_i[X] \ ^{i\in\{1\ldots n\}}])$
$Z' \equiv Obj(X.[l_i \nu_i\colon T'_i[X] \ ^{i\in\{1\ldots n+m\}}])$

$$\frac{\Gamma, X_1, X_2\colon Type, X_1 \subseteq X_2 \vdash T_i[X_1/X] \subseteq T_i[X_2/X] \quad \text{for } i \in \{1\ldots n\}}{\Gamma \vdash Z \in Type} \ \textbf{Obj wf}$$

$$\frac{\Gamma \vdash Z \in Type \quad \Gamma, T \preceq [l_i \nu_i\colon T_i[T/X] \ ^{i\in\{1\ldots n\}}], o\colon T \vdash t_i[o/x_i] \in T_i[T/X]}{\Gamma \vdash z \in Z} \ \textbf{Obj mem}$$

$$\frac{\Gamma \vdash o \in Z}{\Gamma \vdash o.l_i \in T_i[Z/X]} \ \textbf{Obj sel}$$

$$\frac{\Gamma \vdash Z \in Type \quad \Gamma, T \preceq [l_i \nu_i\colon T_i[T/X] \ ^{i\in\{1\ldots n\}}], o\colon T \vdash t[o/x] \in T_i[T/X]}{\Gamma \vdash o\,[l_i \leftarrow \lambda x.t] \in Z} \ \textbf{Obj ovr}$$

$$\frac{\Gamma \vdash Z \in Type \quad \Gamma \vdash Z' \in Type \quad \Gamma \vdash T'_i[Z']^{\nu'_i} \lhd T_i[Z]^{\nu_i} \text{ for } i \in \{1\ldots n\}}{\Gamma \vdash Z' \preceq Z} \ \textbf{Obj sub}$$

$$\frac{\Gamma \vdash S = T \in Type}{\Gamma \vdash S^{\circ} \lhd T^{\circ}} \ \textbf{Sub Inv} \qquad \frac{\Gamma \vdash S \preceq T \text{ for } \nu \in \{^{\circ}, ^{+}\}}{\Gamma \vdash S^{\nu} \lhd T^{+}} \ \textbf{Sub Cov}$$

Figure 11.2: Rules for nondependent calculus

The typing of this object is problematic because the *fact* method uses itself, and it needs its own type in order to derive a type. A non-recursive interpretation can be derived from the following observation: method polymorphism is *class-based*. Once an object is instantiated, method use is monomorphic, with the type fixed at the type of the object. Methods are polymorphic, but recursive calls are *monomorphic*.

In the factorial point example, the recursive use of *fact* is at the type of the argument o, and the *fact* method can be given the following type:

$$\mathit{fact}\colon \bigcap_k T \preceq_{P_F} \{x\colon \{\} \to \mathbb{Z}; \mathit{fact}\colon T \to T\}.T \to T.$$

The interpretation of the object type T is *partial* to account for possible divergence of the *fact* method. Until now, the interpretation has not distinguished between partial and total objects. However, with the introduction of recursive methods, a formal account of totality would require a specification of all the intermediate computations. Our account of partial objects is based on the partial types of Smith [Smi89], which have been simplified by Crary [Crang]. A partial type, denoted $\overline{T}$, contains a term x if the evaluation of x diverges, or if x is an element of T. The induction principle for partial types relies on an *admissiblity* condition, which holds for the types in our calculus. In the rest of the section, we interpret all types as partial.

The interpretation of objects is given in terms of its monomorphic and polymorphic translations. The monomorphic interpretation is used for the object specification in recursive calls. For an object $T = \mathit{Obj}(X.[l_i \nu_i\colon T_i[X]^{\,i\in\{1\ldots n\}}])$ the interpretation is given with respect to an object type O:

$$[\![T]\!]^O_{\mathit{mono}} \equiv \{l_i\colon O \to T_i[O/X]^{\,i\in\{1\ldots n\}}\}.$$

The general interpretation requires that methods also be polymorphic over all subobjects of the monomorphic interpretation.

$$[\![T]\!] \equiv \left\{l_i\colon \bigcap_k O \preceq_T [\![T]\!]^O_{\mathit{mono}}.O \to T_i[O/X]^{\,i\in\{1\ldots n\}}\right\}$$

For example, when applied to the point example P_F, these definitions give the following interpretations:

$$[\![P_F]\!]^O_{\mathit{mono}} \equiv \{x\colon O \to \mathbb{Z}; \mathit{fact}\colon O \to O\}$$

and

$$[\![P_F]\!] \equiv \left\{ \begin{array}{l} x\colon \bigcap_k O \preceq_{P_F} [\![P_F]\!]^O_{\mathit{mono}}.O \to \mathbb{Z}; \\ \mathit{move}\colon \bigcap_k O \preceq_{P_F} [\![P_F]\!]^O_{\mathit{mono}}.O \to O \end{array} \right\}.$$

To complete the example, the relation $O \preceq_{P_F} [\![P_F]\!]^O_{mono}$ has the following definition:

$$\begin{aligned} &O \preceq_{P_F} [\![P_F]\!]^O_{mono} \equiv \\ &\quad \exists R\colon Atom \to Type. \\ &\qquad O = l\colon Atom \to R(l) \\ &\quad \wedge\ R(x) \subseteq O \to \mathbb{Z} \wedge R(x) \supseteq O \to \mathbb{Z} \\ &\quad \wedge\ R(fact) \subseteq O \to O \end{aligned}$$

The types in the the point definitions are considered to be partial, to allow for points with nonterminating methods, and the relation $\preceq_{P_F}$ constrains the polymorphism to subobjects that are invariant in their x method (to allow for updating), and covariant in their *fact* method.

Using the definitions for the general object interpretations, the rules of the object calculus can be derived from the rules of the type theory. The rules for the nondependent object calculus are listed in Figure 11.2. These rules justify the "standard" rules given in Abadi et. al. for the object calculus. The relation $\lhd$ is defined to simplify the rules for subtyping, and is defined in the type theory as follows:

$$\nu_1 T_1 \lhd \nu_2 T_2 \equiv \begin{cases} T_1 = T_2 \in Type & \text{if } \nu_1 =^\circ, \nu_2 =^\circ \\ T_1 \preceq T_2 & \text{if } \nu_1 \in \{^\circ, ^+\}, \nu_2 =^+ \\ False & \text{otherwise} \end{cases}$$

The soundness of these rules have been verified in the Nuprl theorem prover. The proof relies critically on method monotonicity, which specifies that condition under which self-application is allowed.

Definition 6 *Monotonicity. A family of types $\mathcal{T}\colon Type \to Type$ is* monotonic *if for any two types T_1 and T_2 where T_1 is a subtype of T_2, then $\mathcal{T}(T_1)$ is a subtype of $\mathcal{T}(T_2)$. Equivalently:*

$$\forall T_1, T_2\colon Type. T_1 \subseteq T_2 \Rightarrow \mathcal{T}(T_1) \subseteq \mathcal{T}(T_2)$$

For the interpretation of objects, we use monotonicity to infer continuity over the intersection type. For the type theory fragment we use, all types are continuous if they are monotone.

Definition 7 *Continuity. A family of types $\mathcal{T}\colon Type \to Type$ is* continuous *relative to a type T if the following relation holds:*

$$\left(\bigcap T' \subseteq T. \mathcal{T}(T')\right) = \mathcal{T}\left(\bigcap T' \subseteq T. T'\right)$$

Theorem 7 *Soundness. The rules for non-dependent objects are sound with respect to the type theoretic model.*

As we mentioned in Section 11.3.2, the critical rule in the verification is the rule for **Obj sel**, which requires a self-application. We outline the critical step of the proof here, which is the typing of the self-application for $o \in [\![Z]\!]$:

$$[\![o]\!](l_i)([\![o]\!]) \in T_i[[\![Z]\!]/X]$$

The interpretation $[\![Z]\!]$ is a record of polymorphic functions. Since the record is monotonic in the types, we can normalize $[\![Z]\!]$ so that all intersections occur outside the record:

$$[\![Z]\!] = \bigcap_k O \preceq_Z [\![Z]\!]^O_{mono}.\{l_i{:}O \to T_i[O/X]^{\ i\in\{1\ldots n\}}\}$$

To simplify the presentation, we make the following definitions:

$$\begin{array}{rcl} O' & \equiv & [\![Z]\!]^O_{mono} \\ \mathcal{T} & \equiv & \lambda O.\{l_i{:}O \to T_i[O/X]^{\ i\in\{1\ldots n\}}\}, \end{array}$$

and

$$[\![Z]\!] = \bigcap_k O \preceq_Z O'.\mathcal{T}(O).$$

Given the continuity of T_i, the statement

$$[\![o]\!](l_i)([\![o]\!]) \in T_i\left[\bigcap_k O \preceq_Z O'.\mathcal{T}(O)./X\right]$$

can be rewritten to the equivalent statement

$$[\![o]\!](l_i)([\![o]\!]) \in \bigcap_k O \preceq_Z O'.T_i\,[\mathcal{T}(O)/X]$$

which is provable using a direct case analysis.

Although there are several technical steps in this argument, the crucial point is semantic: method monotonicity preserves the predicativity of type intersections. In turn, the type intersections are crucial for describing polymorphism, which provides the basis for two fundamental features of object-oriented languages: method subsumption and inheritance. An extension of this interpretation to binary methods would need to answer the following question: is there a predicative interpretation of type intersections of non-monotonic type families?

While the semantic argument is useful for understanding the mathematics of inheritance, the proof-theory shows how monotonicity is used to restrict proofs to cases with well-known behavior.

Several results follow directly from the interpretation. Our interpretation of objects in the type type theory inherits properties from each of

$$\begin{array}{lrcl@{\qquad}lcl}
\text{Let} & z & \equiv & [l_i = \lambda x_i.T_i \ ^{i\in\{1\ldots n\}}] & & & \\
& Z & \equiv & Obj(X.[l_i\nu_i\colon T_i[X] \ ^{i\in\{1\ldots n\}}]) & Z_i & \equiv & Obj(X.[l_j\nu_j\colon T_j[X] \ ^{j\in\{1\ldots i\}}]) \\
& Z' & \equiv & Obj(X.[l_i\nu_i\colon T'_i[X] \ ^{i\in\{1\ldots n+m\}}]) & Z'_i & \equiv & Obj(X.[l_j\nu_j\colon T'_j[X] \ ^{j\in\{1\ldots i\}}])
\end{array}$$

$$\frac{\Gamma, o\colon Z_{i-1}, X_1, X_2\colon Type, X_1 \subseteq X_2 \vdash T_i[o/\bar{l}, X_1/X] \subseteq T_i[o/\bar{l}, X_2/X] \quad \text{for } i \in \{1\ldots n\}}{\Gamma \vdash Z \in Type} \ \textbf{Obj wf}$$

$$\frac{\Gamma \vdash Z \in Type \quad \Gamma, T \preceq Z_{i-1}, o\colon T \vdash t_i[o/x_i] \in T_i[o/\bar{l}, T/X]}{\Gamma \vdash z \in Z} \ \textbf{Obj mem}$$

$$\frac{\Gamma \vdash o \in Z}{\Gamma \vdash o.l_i \in T_i[\bar{l}, Z/X]} \ \textbf{Obj sel}$$

$$\frac{\Gamma \vdash Z \in Type \quad \Gamma, T \preceq Z_{i-1}, o\colon T \vdash t[o/x] \in T_i[o/\bar{l}, T/X]}{\Gamma \vdash o\,[l_i \leftarrow \lambda x.t] \in Z} \ \textbf{Obj ovr}$$

$$\frac{\begin{array}{l}\Gamma \vdash Z \in Type \\ \Gamma \vdash Z' \in Type \\ \Gamma, o\colon Z'_{i-1} \vdash T'_i[o/\bar{l}, Z'_{i-1}/X]^{\nu'_i} \lhd T_i[o/\bar{l}, Z_{i-1}/X]^{\nu_i} \text{ for } i \in \{1\ldots n\}\end{array}}{\Gamma \vdash Z' \preceq Z} \ \textbf{Obj sub}$$

$$\frac{\Gamma \vdash S = T \in Type}{\Gamma \vdash S^\circ \lhd T^\circ} \ \textbf{Sub Inv} \qquad \frac{\Gamma \vdash S \preceq T \text{ for } \nu \in \{^\circ, ^+\}}{\Gamma \vdash S^\nu \lhd T^+} \ \textbf{Sub Cov}$$

Figure 11.3: Rules for dependent calculus

its models. Progress, preservation, and substitution properties, which normally require proof for object systems, arise directly from a semantics of *functionality* for the type theory. In addition, the type theoretic interpretation wraps the object system in a mathematical framework, making it suitable as a basis for inheritance of *proofs*, the subject of the next section.

11.4 Dependent Objects

The interpretation of objects can also be extended with *dependent* object types, which can be used for defining classes that contain formal specifications (and objects with *proofs*). There are two main issues that

arise in the interpretation:

1. Methods form groups in which all methods must be updated simultaneously to preserve the typing properties.

2. The interpretation of types relies on *dependent* record types, which are defined as *very*-dependent function types.

Dependent method types use Curry-Howard isomorphism to assert propositions about the implementation, and the method implementations are proofs that the specification is satisfied.

In the dependent object calculus, we require that the type dependencies in object types be well-ordered. For object types, we keep the familiar notation

$$Obj(X.[l_i = T_i[X]^{\,i\in\{1\ldots n\}}]),$$

and we allow the types $T_i[X]$ to contain occurrences of the labels $l_1, \ldots, l_{i-1}$, which represent the values of the corresponding methods. For instance, the quantification $\exists x\colon A.B_x$ is isomorphic to the object type $Obj(X.[x\colon A; p\colon B_x])$.

In this section, we extend the P example to two dimensional points on the unit circle:

$$P_{2D} \equiv Obj(X.[x, y\colon \mathbb{R}; p\colon x^2 + y^2 \approx 1; rot : \mathbb{R} \to X])$$

The p method is a *proposition* that constraints the coordinates x and y to lie on the unit circle. The *rot* method is intended to rotate the point by a given angle. Note that the value for x can't normally be updated without also updating the value for y. In general, methods form groups (in this case $\{x, y\}$) in which all of the methods must be updated simultaneously to preserve the specification.

Since methods are required to update members of dependency groups simultaneously, it is reasonable to constrain the polymorphism of methods directly. Rather than using variance annotations on the object methods, we shift the annotations to the polymorphic quantification within the method types. For example, for the P_{2D}, we use the typing

$$rot\colon \bigcap\nolimits_k T \preceq_d Obj(X.[x^\circ, y^\circ\colon \mathbb{R}; p^+\colon x^2 + y^2 \approx 1]).T \to \mathbb{R} \to T.$$

This specification uses an even more constrained subtyping relation $\preceq_d$ that we discuss shortly, and the quantification is over all the methods and constraints that are to be updated.

11.4.1 Encoding

We encode dependent object types as *dependent* records, which in turn are encoded as *very-dependent* function types. Very-dependent function types allow function values to be used to specify their range, in a well-ordered manner. We use the notation

$$\{f \mid x{:}A \to B[f,x]\}$$

to specify the functions f with domain type A, and range type $B[f,a]$ on argument $a \in A$. The domain A must be well-ordered, and $B[f,a]$ may call f only on values $a' \in A$ that are *smaller* than a. For objects and records, the domain type A is the type of labels *Atom*. We always write objects so that the label order is the well-order we are using over the domain *Atom* (smaller elements first).

The encoding of dependent records as very-dependent functions is as follows. Given a dependent record type $R = \{l_i{:}T_i\ ^{i\in\{1\ldots n\}}\}$, where the labels $l_1,\ldots,l_{i-1}$ may occur in the types T_i, we have the encoding:

$$R \equiv \left\{ \begin{array}{ll} f \mid l{:}\mathit{Atom}.\ \mathbf{case}\ l\ \mathbf{of} & \\ \qquad\qquad \mid \cdots & \\ \qquad\qquad \mid l_i{:}T_i[f(l_j)/l_j]^{j\in\{1\ldots i-1\}} & \\ \qquad\qquad \mid \cdots & \\ \qquad\qquad \mid _{:}\mathit{Top} & \end{array} \right\}$$

We modify object types so that the methods specify the method groups they are selecting and updating adjacent to their method label. The general form is:

$$\mathit{Obj}(X.[l_i(\overline{a_i}){:}B_i[X]\ ^{i\in\{1\ldots n\}}]),$$

where $\overline{a_i}$ is a vector of annotations $l_j{:}v_j$. We typically leave out the annotations if the vector $\overline{a_i}$ is empty. For instance, the fully annotated P_{2D} type is

$$P_{2D} \equiv \mathit{Obj}\left(X.\left[\begin{array}{l} x,y{:}\mathbb{R}; p{:}x^2+y^2\approx 1; \\ \mathit{rot}(x^\circ,y^\circ,p^+){:}\mathbb{R}\to X \end{array}\right]\right),$$

which states that the *rot* method updates x and y, and it refers to p (perhaps implicitly). Note that the type

$$P_{2D} \equiv \mathit{Obj}(X.[x,y{:}\mathbb{R}; p{:}x^2+y^2\approx 1; \mathit{rot}(x^\circ,p^+){:}\mathbb{R}\to X])$$

is a valid type, but it does not permit the *rot* method to update the value

of $\mathcal{Y}$.

The encoding of object types is similar to the nondependent case. Consider an object type

$$O \equiv Obj(X.[l_i(\overline{a_i}): T_i[X]\ ^{i\in\{1\ldots n\}}])$$

that does not contain recursive method, where

$$\overline{a_i} \equiv a_{i_1}: \nu_{i_1}, \ldots, a_{i_m}: \nu_{i_m}.$$

The encoding for O is defined as follows:

$$\begin{array}{rcl} [\![T_0]\!] & = & \{\} \\ [\![T_{i+1}]\!] & = & [\![T_i]\!] \cap \left\{ l_{i+1}: \bigcap_k T \preceq_d [\![T_i]\!].T \to T_{i+1}[T/X] \right\} \end{array}$$

and $[\![O]\!] = [\![T_n]\!]$.

11.4.2 Dependent subtyping: $\preceq^{dep}$

The dependent encoding is essentially the same as the nondependent case, except for the occurrence of the $\preceq_d$ relation, and the fact that each method has its own variance annotations. The $\preceq_d$ relation is intended to prevent dependencies from adversely constraining invariant types. For example, the type $[x^\circ: \mathbb{Z}; p: x = 0 \in \mathbb{Z}]$ constrains the type for x, even though it has the same method type for x as the object type $[x^\circ: \mathbb{Z}]$. The $\preceq_d$ prevents this scenario by explicitly stating that invariant elements are overridable. Let $T \equiv Obj(X.[l_i \nu_i: T_i[X]\ ^{i\in\{1\ldots m\}}])$, and let *upd* be the collection of indices of invariant methods in T, and let L_T^{upd} be the corresponding collection of labels.

$$\begin{array}{rcl} T' \preceq_d T \equiv & T' \subseteq T & \\ & \wedge\ \forall t: T'.\forall z: Obj(X.[l_i: T_i\ ^{i\in upd}]).\exists t': T. & \\ & \quad t' = t \in Atom/L_T^{upd} \to Type & \\ & \wedge\ t' = z \in L_T^{upd} \to Type & \end{array}$$

This definition states explicitly that the subtypes under consideration are only those where values for the invariant methods can be updated with any values that are mutually consistent. Given this definition, we can derive the rules shown in Figure 11.3 for the dependent object calculus. In the table, we use the notation $T_i[o/\bar{l}]$ to mean the substitution $T_i[o.l/l]$ for each $l \in \bar{l}$ where $\bar{l}$ is the set of all labels in T_i.

Theorem 8 *The rules for dependent objects are sound.*

The proof of soundness is essentially the same as for the nondependent-rules, except to account for the additional bindings in the method types.

As before, the type theory justifies standard properties for the object calculus. The functionality of the type theory guarantees progress, preservation, and substitution properties.

11.5 Related Work

Another framework for expressive object calculi is developed by Hofmann et el. [HNSS98], who have implemented a verification calculus based on the existential interpretation of objects. In their system, objects have three parts: an object state, the object methods, and the proofs of method correctness (dependent objects with exactly one dependency). Their interpretation uses an existential encoding of objects [PT94, HP95b] in an impredicative type theory. Although the existential encoding restricts method update, the interpretation of objects does not require the use of recursive types, allowing the development of expressive type systems [CP96, HP95a].

The interpretation of object relies on a restricted subtyping relation (we use notation $\preceq$) to describe valid sub-objects. This is important because the general subtype relation would disallow updates. Our restricted subtyping is essentially the same as the *positive subtyping* of Hofmann and Pierce [HP95a]. In the definition of restricted subtyping, we describe which parts of the type are not allowed to change. Hofmann and Pierce describe it in terms of coercions: $S \preceq T$ if there is a coercion $S \to T$, and there is a function in $S \to T \to S$ that replaces the "T" part of S with a new element of T. In their account, Hofmann and Pierce use the subtyping to define an existential encoding of objects in $F^{\omega}_{\leq}$..

Another existential system was developed by Jackson [Jac95] to formalize a significant portion of constructive algebra in Nuprl. While Jackson's framework includes many of the properties of objects, the subtyping requirements prohibit the use of self-application and constrain the shape of objects. Our work has borrowed from Jackson' interpretation; in an earlier paper [Hic96], we presented another existential interpretation of objects in a predicative type theory, using the very dependent function type to express the type dependencies.

THE METAPRL LOGICAL PROGRAMMING ENVIRONMENT

Volume II

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Jason Jonathan Hickey

January 2001

Part V

Logics Implemented in MetaPRL

Chapter 12

The MetaPRL Logics

This is Volume II (Part V) on the MetaPRL system. This book describes and documents the logics defined in MetaPRL. There are four logics.

The "base" theory (Chapter 12) The "base" theory implements basic tactics, tacticals, conversions (the rewriting package), and conversionals. The base theory also implements the infrastructure for building the more advanced tactics, like the `autoT` and `dT` tactics, as well as the conversion `reduceC` for term evaluation.

A first-order logic (Chapter 13) The `Fol` theory defines both a constructive and a classical first-order logic. This chapter is presented in the form of a **tutorial**.

The Nuprl type theory (Chapter 14) The `Itt_logic` module defines the Nuprl type theory.

Aczel's constructive CZF set theory (Chapter 15) The CZF set theory is a constructive version of ZF set theory. This logic provides an example of a large derivation (of CZF) from the Nuprl type theory.

It is expected that the reader will understand the basic principles of the MetaPRL system, covered in Book I, especially Chapter 7, which documents the module structure and the formal objects that are used to define theories.

Except for the chapter on first-order logic, this documentation is generated from the actual source text of the logics defined in MetaPRL. MetaPRL uses a form of literate programming, where the documentation for a logic is interleaved with the definition of the logic. This has several consequences. Since the documentation is not separate from the logical definition, it is more likely that the documentation will be maintained in an accurate state. Conversely, the documentation contains the *actual* logical definitions, so it is less likely to contain a transciption error.

Errors are still possible—when the readable documentation is produced from the source text, the program text is formatted by the system using display forms (Section 7.3), which may omit some of the logical detail.

As discussed in the chapter on the module system (Chapter 7), MetaPRL logics are defined in a inheritance hierarchy. All logics share a common root, the `Base_theory`, which defines common forms for automation and display. The Nuprl type theory inherits the `Base_theory`, and the first-order logic and CZF inherit the type theory.

12.1 Base_theory module

The `Base_theory` "theory" is not a theory in the strict sense. It defines only two rules (in the `Base_rewrite` module; Section 12.10). Instead, it serves as the connection to the primitive MetaPRL prover, and it defines several resources that are useful for defining tactics.

The connection with the primitive MetaPRL prover is defined in the modules `Top_tacticals` (Section 12.5) and `Top_conversionals` (Section 12.6). The `Top_tacticals` module defines the primitive tactics (such as the identity and failure tactics), and several tacticals for tactic composition (such as the `thenT` and `orelseT` tacticals). The `Top_conversionals` module defines *conversions*, which play the role of tactics for *rewriting*.

The base theory also defines several resources that can be used to build automated tactics. The `Mptop` module (Section 12.3) defines a resource for adding functions to the MetaPRL interactive top-loop. The `Base_auto_tactic` module (Section 12.8) defines a resource for defining the `trivialT` and `autoT` tactics for automated reasoning. The `Base_dtactic` module (Section 12.9) defines resources for adding introduction and elimination reasoning to the `dT` tactic.

The `Base_theory` module itself defines no new primitives; it simply bring together all the parts of the base logic into a unified module. Descendent theories (such as the Nuprl type theory, Section 14.32, and the first-order logic Section 13) include the `Base_theory` directly.

12.1.1 Parents

include Summary
include Mptop
include Var
include Top_tacticals

include Top_conversionals
include Base_trivial
include Base_auto_tactic
include Base_dtactic
include Base_rewrite

12.2 Summary module

The `Summary` module implements display forms for top level module summaries. A *summary* is a term representation of a top-level module. In most cases, MetaPRL produces the summary when the ML file corresponding to the module is compiled. The term contains *all* the information in the original file, and it also contains terms that represent the proof structure of the module.

The term representation of the summary is used to represent the module in the logical library. It is used by the MetaPRL editor to display the module so that it can be browsed. Various parts of the summary, such as terms representing rules, proofs, are used during proof editing to display the current node and goal in a proof.

The `Summary` module is built-in: the summary term is constructed by the MetaPRL compiler. This module is the connection to the compiler, and its purpose is to allow the inner data structures of the compiler to be browsed and edited.

The file contains a declaration for each of the types of term produced by the compiler, and it also contains display definitions to display the terms in a natural way. In some cases, such as for LaTeX and HTML mode, special display forms are used to produce code to display the terms correctly. The display forms can be quite complex; we do not include their description in this document.

12.2.1 Terms

There are two outermost terms for modules: the interface is wrapped with `interface` term, and the implementation is wrapped with `implementation`.

declare `interface` = Summary!interface{'intf}
declare `implementation` = Summary!implementation{'impl}

The `comment` term is used to enclose structured comments (term comments).

declare `comment` = Summary!comment{'t}

Each of the components in an interface or implementation also has a term description. The `parent` term describes an **include** statement: the *path* is the pathname of the parent module; the *opens* is the complete list of parents (the transitive closure) and the *resources* are the complete list of resources defined or used by the module.

The `opname` term described a **declare** statement. The *term* is the term being declared, and the *name* is the opname of the declaration.

```
declare parent = Summary!parent{'path; 'opens; 'resources}
declare opname = Summary!opname[name:s]{'term}
```

Rewrites are defined with the `rewrite` and `cond_rewrite` terms. The *redex* and *contractum* define the rewrite; the *proof* is the proof of the rewrite (which is empty if the rewrite is primitive). for the conditional rewrite the *params* and *args* are the terms the defined the assumptions under which the rewrite is valid. The *name* is the name of the rewrite.

```
declare rewrite
    = Summary!rewrite[name:s]{'redex; 'contractum; 'proof}
declare cond_rewrite
    = Summary!cond_rewrite
            [name:s]
            {'params;
             'args;
             'redex;
             'contractum;
             'proof}
```

Rules are defined using the `axiom` and `rule` terms. An axiom is a simple rule with no assumptions; *stmt* is the statement of the axiom, and *proof* is it's proof. The rule also includes a *param* list that defines the subgoals of the rule.

```
declare axiom = Summary!axiom[name:s]{'stmt; 'proof}
declare rule = Summary!rule[name:s]{'params; 'stmt; 'proof}
```

Resources are declared with the `resource` term, which has three subterms corresponding to the three types in the resource definition. The *extract* is the type of values provided by the resource, *data* is the internal type used to represent the resource data, and *improve* is the type of arguments that are required to make additions to the resource.

```
declare resource
    = Summary!resource[name:s]{'extract; 'improve; 'data}
declare resource_defs = Summary!resource_defs{'res}
```

Infix definitions (like the tacticals `thenT` and `orelseT`) are defined with the `infix` declaration.

OCaml definitions are also represented as terms using the `summary_item` term, where the *term* is the term that represents the OCaml code.

declare `infix` = Summary!infix[name:s]
declare `summary_item` = Summary!summary_item{'term}

Display forms are represented using the `dform` term. The *modes* are the valid modes of the display form (for example, the normal display mode "prl", or the "html" or "tex" display modes), as well as parenthesization and precedence definitions. The *redex* is the term to be displayed, and the *def* is the term describing the displayed definition.

declare `dform` = Summary!dform{'modes; 'redex; 'def}
declare `prec` = Summary!prec[name:s]
declare `prec_rel` = Summary!prec_rel[pr:s]
declare `inherit_df` = Summary!inherit_df
declare `prec_df` = Summary!prec_df[name:s]
declare `parens_df` = Summary!parens_df
declare `mode_df` = Summary!mode_df[mode:s]
declare `except_mode_df` = Summary!except_mode_df[mode:s]

Rules and axioms are described with *meta*-terms. The meta-terms are defined inductively: a term (a `meta_theorem`) is a meta-term, and given two meta-terms A and B, so are the meta-implication `meta_implies`, the meta-bi-implication `meta_iff`, and the dependent meta-function `meta_function` where *arg* is a variable quantified over A and bound in B. The `meta_labeled` term is used to add a label to a meta-term.

declare `meta_theorem` = Summary!meta_theorem{'A}
declare `meta_implies` = Summary!meta_implies{'A; 'B}
declare `meta_function` = Summary!meta_function{'arg; 'A; 'B}
declare `meta_iff` = Summary!meta_iff{'A; 'B}
declare `meta_labeled` = Summary!meta_labeled[label:s]{'meta}
declare `meta_theory` = Summary!meta_theory{'A}

A proof has a goal define with the `goal` term, where the *status* is the status of the proof, the `label` is the label of the outermost proof node, the *assums* are the assumptions (the subgoals) of the theorem being proved, and the *goal* is the goal term.

The status of a proof can have four values: a proof is *bad* if it can be shown that the proof is inconsistent; it is *partial* if it has unproven subgoals; it is *asserted* if it is primitive or if it has not been checked; and it is *complete* if it has been checked.

declare goal = Summary!goal{'status; 'label; 'assums; 'goal}
declare status_bad = Summary!status_bad
declare status_partial = Summary!status_partial
declare status_asserted = Summary!status_asserted
declare status_complete = Summary!status_complete
declare status = Summary!status{'sl}

12.3 Mptop module

The Mptop module defines a simplified OCaml top-loop that is used by the MetaPRL editor to evaluate user input. The evaluator handle only a few basic types (for example, for strings, numbers, terms, and tactics), and it handle function application. It does not implement more sophisticated OCaml expressions such as function definition and pattern matching.

12.3.1 Parents

include Summary

The valid expression types are given with the following type definition.

```
type expr =
    (* Base types *
    UnitExpr of unit
  | BoolExpr of bool
  | IntExpr of int
  | StringExpr of string
  | TermExpr of term
  | TacticExpr of tactic
  | ConvExpr of conv
  | AddressExpr of address

    (* Untyped tuples and functions *
  | ListExpr of expr list
  | TupleExpr of expr list
  | FunExpr of (expr -> expr)

    (* Common cases are typed *
```

```
  | UnitFunExpr of (unit -> expr)
  | BoolFunExpr of (bool -> expr)
  | IntFunExpr of (int -> expr)
  | StringFunExpr of (string -> expr)
  | TermFunExpr of (term -> expr)
  | TacticFunExpr of (tactic -> expr)
  | IntTacticFunExpr of ((int -> tactic) -> expr)
  | ConvFunExpr of (conv -> expr)
  | AddressFunExpr of (address -> expr)

    (* These functions take lists *
  | AddrFunExpr of (int list -> expr)
  | StringListFunExpr of (string list -> expr)
  | TermListFunExpr of (term list -> expr)
  | TacticListFunExpr of (tactic list -> expr)
  | ConvListFunExpr of (conv list -> expr)
```

Toplevel values are added to the `toploop_resource` resource. The argument has type `string * expr`, which includes the string name of the value, and it's value.

12.4 Var module

The `Var` module provides utilities to generate new variables that are guaranteed to be distinct from all other bound variables in a proof goal. For example, the `productElimination` (Section 14.15) rule, splits a hypothesis of the form $x: T_1 \times T_2$ into two hypotheses $u: T_1$ and $v: T_2$. The variables u and v have to be chosen at rule application time, and this module assists in the generation of new names.

There are three basic functions implemented here.

```
val new_var         : string -> string list -> string
val maybe_new_var   : string -> string list -> string
val maybe_new_vars  :
   string list -> string list -> string list
```

The function $\texttt{new}_{\texttt{var}}\ v\ \textit{vars}$ generates a new variable "similar" to v, but not contained in *vars*. In this case "similar" means that the variable has the same name, but it may have a numerical suffix to make it distinct.

12.4.1 Parents

include Summary

12.5 Top_tacticals module

The `Top_tacticals` module defines the primitive tactics and tacticals provided by the MetaPRL prover.

12.5.1 Parents

include Mptop

12.5.2 Primitive tactics

`idT` The `idT` tactic is the *identity*.

$$\begin{array}{l} [ext]\ H \vdash T \\ \text{—BY idT} \\ [ext]\ H \vdash T \end{array}$$

`cutT` The `cutT` tactic implements primitive lemma-instantiation.

$$\begin{array}{l} [ext]\ H \vdash T_1 \\ [ext]\ H; x\colon T_1 \vdash T_2 \\ \text{—BY cutT}\ T_1 \\ [ext]\ H \vdash T_2 \end{array}$$

`failT`, `failwithT` The `failT` tactic always fails, and the `failWithT` fails with a specific message.

`nthAssumT` The `nthAssumT` tactic proves a goal by *assumption*. Technically, an assumption is a subgoal of the theorem being proved. The assumption must be syntactically identical to the goal.

$$\begin{array}{l}
\| \; . \\
\|\text{—}\texttt{BY nthAssumT}i \\
\| \; [ext]\ H_1 \vdash T_1 (\textit{Assumption 1}) \\
\| \; \ldots \\
\| \; [ext]\ H_i \vdash T_i (\textit{Assumption } i) \\
\| \; \ldots \\
\| \; [ext]\ H_n \vdash T_n (\textit{Assumption } n) \\
\| \; \hline
\| \; [ext]\ H_i \vdash T_i
\end{array}$$

12.5.3 Tacticals

`thenT`, `orelseT` There are several tacticals to manage proof search. The basic tacticals are `thenT` and `orelseT`, both *infix* tacticals. The tactic tac_1 `thenT` tac_2 first applies tac_1 to the goal, and then applies tac_2 to *all* of the subgoals. It fails if either tactic fails on any of the subgoals. The tactic tac_1 `orelseT` tac_2 first applies tac_1. If it succeeds, tac_2 is ignored. Otherwise, tac_2 is applied to the original goal.

`firstT` The `firstT` tactical is a variant of `orelseT`. It takes a list of tactics $[tac_1; \ldots; tac_n]$ to be applied in order until the first one succeeds. It fails if all of the argument tactics fail. It is equivalent to (tac_1 `orelseT` $\cdots$ `orelseT` tac_n).

`seqT`, `seqOnSameConclT` The `seqT` is the universal form of the `firstT` tactical. The (`seqT` $[tac_1; \ldots; tac_n]$) tactic is equivalent to (tac_1 `thenT` $\cdots$ `thenT` tac_n). The `seqOnSameConclT` tactic is the same as `seqT` except that is selects only those subgoals that have the same conclusion as the current goal.

`then_OnFirstT`, `then_OnLastT`, `then_OnSameConclT` There are also several tactics that apply to selected subgoals by number. The use of these tactics is discouraged in favor of selecting tactics by label (discussed below).

The tac_1 `then_OnFirstT` tac_2 applies tac_1 to the goal, and then applies tac_2 to the *first* subgoal that is generated. The `then_OnLastT` selects the last subgoal, and the `then_OnSameConclT` tactic chooses the subgoal with the same conclusion as the current goal.

`timingT` The `timingT` tactical applies its argument, and prints timing information useful for tactic profiling.

```
|| [ext] H ⊢ T
|—BY idT
|| [ext] H ⊢ T
```

```
User time 0.000000; System time 0.000000
```

`tryT`, `completeT` The `tryT` tactical applies its argument, and performs the identity if the tactic fails. The tactic (`tryT` *tac*) is equivalent to (*tac* `orelseT idT`).

`progressT` The (`progressT` *tac*) tactic applies it argument and fails if either 1) *tac* fails, or *tac* failed to make "progress" by generating subgoals that differ from the current goal.

`repeatT`, `repeatForT` The (`repeatT` *tac*) tactic applies the argument *tac* repeatedly to the current goal and all of the generated subgoals until 1) an application of *tac* fails, or 2) the proof is complete (there are no subgoals). The (`repeatForT` i *tac*) repeats the application at most i times.

12.5.4 Tactic arguments

Tactics and rules may require arguments of various types (such as numbers, strings, terms, or even other tactics). The proof structure allows the proof tree to be annotated with these arguments, and the arguments are inherited down the proof tree until they are removed. This allows the proof to act as a temporary environment for communication between tactics. The following tactics apply temporary annotations.

`withT`, `withTypeT`, `withBoolT`, `withIntT` The (`withT` *term tac*) tactic adds a term annotation to the tree; applies *tac*; and then removes the term annotation from all of the subgoals. The `withTypeT` adds a "type" annotation (the type is expressed as a term). The `withBoolT` and `withIntT` add Boolean and integer annotations.

`selT` There is one more tactical that is frequently used in the MetaPRL logics: by convention the `selT` tactical is used to "select" among several alternate methods of proof. For example, in proving a disjunction (Section 14.17) it is necessary to select the branch of the disjunct.

```
|| [ext] H; x: T2; J ⊢ T1 Type
|| [ext] H; x: T2; J ⊢ T2
|—BY selT2(dT0)
|| [ext] H; x: T2; J ⊢ T1 ∨ T2
```

12.5.5 Clause selection

The following tactics are intended for use in a single-conclusion sequent calculus. A sequent $[ext]\ x_1{:}T_1; \cdots; x_n{:}T_n \vdash C$ has $n+1$ *clauses*. The hypotheses are clauses $1, \ldots, n$ and the conclusion is clause 0.

`onClauseT`, `onHypT`, `onConclT` The (`onClauseT` *i tac*) tactic applies the argument tactic with integer argument i (it is equivalent to *tac i*). The `onHypT` restricts the integer argument to be a valid hypothesis number, and the (`onConclT` *tac*) tactical applies its argument tactic with argument 0.

`onVarT` The (`onVarT` *tac*) applies its argument with the number of the hypothesis labeled with variable v.

`onClausesT`, `onHypsT` The `onClausesT` and `onHypsT` take a list of clause numbers. The (`onClausesT` $[i_1; \cdots; i_n]$ *tac*) is equivalent to (*tac* i_1 `thenT` $\cdots$ `thenT` *tac* i_n `thenT` *tac* 0). The `onHypsT` does the same, but it requires that the indices correspond to valid hypothesis numbers.

`onAllClausesT`, `onAllHypsT` The (`onAllClausesT` *tac*) applies the argument tactic to all the clauses, including all the hypothesis and the conclusion. The `onHypsT` applies the argument only to the hypotheses.

`onSomeHypT` The (`onSomeHypT` *tac*) applies the argument tactic to the hypotheses from the last to the first, returning once an application succeeds. The (`onSomeHypT` *tac*) is equivalent to (*tac* n `orelseT` $\cdots$ `orelseT` *tac* 1).

12.5.6 Labels

Each node in a proof tree has a *label*. The labels have no logical meaning, but they are frequently used to provide an informal description of the kind of subgoal. A label can be any string, but there are three commonly used labels: "main" identifies the main steps of a proof, "antecedent" identifies nodes that are assumptions that have to be proved, and "wf" identifies nodes that require well-formedness reasoning.

`addHiddenLabelT`, `removeHiddenLabelT`, `keepingLabelT` There are three tacticals that directly manipulate the label. The (`addHiddenLabelT` "label") tactic assigns the label to the current goal, the `removeHiddenLabelT` tactic assigns the label "main", and the (`keepingLabelT` *tac*) applies the tactic *tac* and assigns

the label of the current goal to all the the remaining subgoals. The "Hidden" is of historical significance only; the labels are hidden only in the sense that they have no logical significance.

ifLabT In addition to manipulating the labels, there are several tacticals that take advantage of the label to apply tactics selectively. The (ifLabT *l tac*) applies the tactic *tac* only if the current goal has label *l*.

thenMT, thenAT, thenWT The (infix) thenMT, thenAT, and thenWT are like the thenT tactical, except that they apply their second argument only to the goals labeled "main", "antecedent", or "wf" respectively.

repeatMT The repeatMT tactic is similar to repeatT, but it applies only to the subgoals labeled "main".

12.6 Top_conversionals module

Conversionals are analog of tactics (Section 12.5) for rewriting. Conversionals are used extensively in the Nuprl type theory (Section 14.32) to express and apply computational equivalences. The Top_conversionals module defines the basic conversionals provided by the MetaPRL prover.

Each **rewrite** definition in a module defines a conversion. For example, the definition of beta reduction in the Nuprl type theory (Section 14.11), is defined as follows:

$$\textbf{rewrite}\ \text{unfold_beta} : (\lambda x.b[x])\ a \longleftrightarrow b[a]$$

This declaration defines a conversion called unfold_beta that can be applied with the function rwh, which searches for the outermost valid applications of the rewrite. Here is an example proof step:

$$\begin{array}{l} [ext]\ H \vdash ((\lambda v.v + 1)\ 2) = 3 \in \mathbb{Z} \\ \text{—BY rwhunfold}_{\text{beta}}\text{0} \\ [ext]\ H \vdash 2 + 1 = 3 \in \mathbb{Z} \end{array}$$

12.6.1 Parents

include Mptop

12.6.2 Conversion application

`rw` Conversions are not tactics: they have a different type `conv` and they are applied differently. The basic method for applying a conversion is to use `rw`, which converts a conversion to a tactic applied to a specific clause in a sequent (these functions are defined only for a sequent calculus). The (`rw` *conv i*) *tactic* applies the conversion *conv* to clause *i* in the current goal sequent.

12.6.3 Primitive conversions

`idC`, `failC` The `idC` conversion is the identity conversion: no rewriting is performed. The `failC` conversion always fails.

12.6.4 Conversionals

`andthenC`, `orelseC` Conversionals can be combined in the same manner as tactics. The (c_1 `andthenC` c_2) conversion first applies conversion c_1, and then applies c_2 to the result term. The (c_1 `orelseC` c_2) conversion first applies c_1; if c_1 fails (because the conversion does not match the term being rewritten, or because of a call to `failC`), c_2 is applied instead.

`tryC`, `firstC` There are several variations on `orelseC`. The (`tryC` *c*) conversion is equivalent to (*c* orelseC idC). The `firstC` conversion takes a list of conversions to try in order until the first one succeeds. The conversion (`firstC` $[c_1;\cdots;c_n]$) is equivalent to c_1 `orelseC` $\cdots$ `orelseC` c_n.

`repeatC` The (`repeatC` *c*) conversion applies conversion *c* repeatedly until it fails, or until it fails to make progress.

12.6.5 Addressing and search

Generally, the terms to be rewritten to not occur at the outermost level of a clause. The following conversionals recursively search through the subterms of a clause for applicable rewrites.

`someSubC` The most general of these is the (`someSubC` *c*) conversion, which tries applying conversion *c* to all of the immediate subterms of the clause. It succeeds if *c* succeeds on any of the subterms; it fails otherwise. The conversion `allSubC` requires success on *all* of the immediate subterms.

`addrC` Subterms can also be addressed explicitly with the (`addrC` *addr* *c*) conversion, although the use is discouraged. The address is an integer list that describes the *path* leading to the term to be rewritten. For example, the address [] is the identity address, [0] is its leftmost subterm, [0;1] is the second subterm of the first subterm, etc. Addresses are fragile, and correct addresses are difficult to discover. For this reason, the `addrC` conversion is almost never used.

`higherC` The (`higherC` *c*) conversion searches for the outermost occurrences of subterms in the clause where conversion *c* applies. It's definition uses `someSubC`.

```
let rec higherC c = c orelseC (someSubC (higherC c))
```

`lowerC`, `sweepDnC` The `lowerC` conversional searches for the *innermost* rewrite occurrences. The (`sweepDnC` *c*) conversion applies *c* from the outermost to the innermost subterms.

```
let rec sweepDnC c =
   (tryC c) andalsoC (someSubC (sweepDnC c))
```

`sweepUpC` The `sweepUpC` conversion works from the innermost to outermost subterms. Note that these conversions never fail; however they may fail to make progress if the conversion *c* never succeeds.

`rwh` For convenience, the `rwh` function automatically applies the `higherC` conversion, the tactic (`rwh` *c* *i*) is equivalent to (`rw (higherC` *c*`)` *i*).

12.6.6 Conversion reversal

Computational rewrites define a congruence, and all of the equivalence relations hold, including reversing the application of the rewrite. However, reversed rewrites are usually incompletely specified.

`foldC`, `cutC` The (`foldC` *t* *c*) takes a term *t* and a conversion that rewrites the term in the *forward* direction, and generates reversed conversion. For example, here is a reverse application of the beta rewrite.

$$\left\| \begin{array}{l} [ext]\ H \vdash 2 + 1 = 3 \in \mathbb{Z} \\ \text{—BY } \mathtt{rwh}(\mathtt{foldC}(\lambda v.v + 1)\ 2\mathtt{unfold}_{\mathtt{beta}})0 \\ [ext]\ H \vdash ((\lambda v.v + 1)\ 2) = 3 \in \mathbb{Z} \end{array} \right.$$

The cutC conversion is used to replace a term and generate a rewrite obligation.

$$\begin{array}{l} [ext]\ H \vdash 3 = 3 \in \mathbb{Z} \\ [ext]\ H \vdash ((\lambda v.v + 1)\ 2) \longleftrightarrow 3 \\ \text{—BY rw(addrC[1](cutC3))0} \\ [ext]\ H \vdash ((\lambda v.v + 1)\ 2) = 3 \in \mathbb{Z} \end{array}$$

12.6.7 Resources

The reduce resource

The reduce resource provides a generic method for defining *evaluation*. The reduceC conversion can be used to apply this evaluator.

For example, the Nuprl type theory describes several generic reductions:

beta $(\lambda v.b[v])\ a \longleftrightarrow b[a]$

pair $(\textbf{match}\ (a,b)\ \textbf{with}\ u,v \to c[u,v]) \longleftrightarrow c[a,b]$

union $(\textbf{match}\ inl(a)\ \textbf{with} inl(u) \to b[u] \mid inr(v) \to c[v]) \longleftrightarrow b[a]$

Each of the modules for functions (Section 14.11), tuples (Section 14.15), and union (Section 14.10), defines an addition to the reduce_resource: the Itt_rfun adds the reduce_beta rewrite with redex $(\lambda v.b[v])\ a$; the Itt_dprod adds the reduceSpread rewrite with redex $(\textbf{match}\ (a,b)\ \textbf{with}\ u,v \to c[u,v])$; and the Itt_union adds the reduceDecideInl rewrite with redex $(\textbf{match}\ inl(a)\ \textbf{with} inl(u) \to b[u] \mid inr(v) \to c[v])$

In modules that include these three theories, the reduceC conversion will recursively search for applications of these three rewrites in an attempt to fully reduce the term.

The implementation of the the reduce_resource and the reduceC conversion rely on tables to store the shape of redices, together with the conversions for the reduction.

12.7 Base_trivial module

Logics that include “trivial” reasoning need a “squash” term to denote proofs that contain no computational content, and an “it” term to denote the proof that has the squash type.

12.7.1 Parents

include Summary

12.8 Base_auto_tactic module

The Base_auto_tactic module defines two of the most useful tactics in the MetaPRL prover. The autoT tactic attempts to prove a goal "automatically," and the trivialT tactic proves goals that are "trivial." Their implementations are surprisingly simple—all of the work in automatic proving is implemented in descendent theories.

This module describes the *generic* implementation of the autoT and trivialT tactics. They are implemented as resources containing collections of tactics that are added by descendent theories. The auto_resource builds collections of tactics specified by a data structure with the following type:

```
type auto_tac = tactic_arg -> (tactic * auto_tac) list

type auto_info =
   { auto_name : string;
     auto_prec : auto_prec;
     auto_tac : auto_tac
   }
```

The auto_name is the name used to describe the entry (for debugging purposes). The auto_tac is a function that takes the current goal (the tactic_arg) and provides a tactic that can be applied to the goal, and a new auto_tac that can be used on the subgoals. The entries are divided into precedence levels; tactics with higher precedence are applied first.

In most cases, the auto_tac is a simple function that just applies a tactic repeatedly. The following function converts a simple tactic into an auto_tac.

```
let rec auto_wrap (tac : tactic) =
   (fun p -> [tac, auto_wrap tac])
```

The more general form of auto_tac (where a new tactic is computed for each goal) is used for tactics that keep track of their context. For instance, a first-order prover may wish to keep a cache of goals that have been examined, and avoid proving the same goal multiple times.

The trivialT is similar to the autoT tactic; it uses the same data structures, but it is used to classify only "trivial" tactics (such as proof by assumption). The separation is made for practical purposes. At times, the autoT tactic may spend significant time performing sophisticated reasoning when a goal has a trivial proof. The trivialT is added to the auto_resource with a high precedence, so "trivial" reasoning is included by default in autoT.

12.8.1 Parents

include Mptop

12.9 Base_dtactic module

The dT tactic is the cornerstone of reasoning in most logics; it provides generic application of introduction elimination reasoning. The Base_dtactic defines a *generic* resource that can be used to add introduction and elimination reasoning. In addition, it add resource *annotations* that can be used in rule definitions to add them automatically to the d resource.

The d resource is implemented as two resources. The Summary!resource

is used to collect introduction rules; and the Summary!resource

is used to collect elimination rules. The components of both resources take a term that describes the shape of the goals to which they apply, and a tactic to use when goals of that form are recognized. The elim_resource takes a tactic of type int -> tactic (the tactic takes the number of the hypothesis to which it applies), and the intro_resource takes a tactic of type tactic.

The (dT *i*) tactic is a generic tactic that takes the clause number of the clause (either hypothesis or conclusion) to "decompose," and it applies the most appropriate entry from the resources.

The resources also allow resource annotations in rule definitions. Typically, the annotation is added to explicit introduction or elimination rules, like the following:

rule and_intro {|intro_resource[]|}H =
$[ext]\ H \vdash A$
$[ext]\ H \vdash B$
$[ext]\ H \vdash A \wedge B$

Once this rule is defined, an application of the tactic (dT 0) to a conjunction will result in an application of the and_intro rule.

The resource annotations take a list of optional arguments. The intro_resource takes arguments of the following type:

```
type intro_option =
   SelectOption of int
 | IntroArgsOption of
      (tactic_arg -> term -> term list) * term option
```

The SelectOption is used for rules that require a selection argument. For instance, the disjunction introduction rule has two forms for the left and right-hand forms.

rule or_intro_left {|intro_resource[$SelectOption1$]|}H =
$[ext]\ H \vdash BType$
$[ext]\ H \vdash A$
$[ext]\ H \vdash A \wedge B$

rule or_intro_right {|intro_resource[$SelectOption2$]|}H =
$[ext]\ H \vdash AType$
$[ext]\ H \vdash B$
$[ext]\ H \vdash A \wedge B$

These options require selT arguments: the left rule is applied with selT 1 (dT 0) and the right rule is applied with selT 2 (dT 0).

The IntroArgsOption is used to *infer* arguments to the rule. The function argument takes the current goal and a subterm, and it provides an argument list for the rule application. The term option entry describes the subterm to be used for the second function argument.

The elim_resource options are defined with the following type:

```
type elim_option =
   ThinOption of (int -> tactic)
 | ElimArgsOption of
     (tactic_arg -> term -> term list) * term option
```

The ElimArgsOption provides arguments in the same way as the IntroArgsOption. The ThinOption is an argument that provides an optional tactic to "thin" the hypothesis after application of the elimination rule.

The dT resources are implemented as tables that store the term descriptions and tactics for "decomposition" reasoning. The dT tactic select the most appropriate rule for a given goal and applies it. The (dT 0) tactic is added to the auto_resource by default.

12.9.1 Parents

include Base_auto_tactic

12.10 Base_rewrite module

The rewrite judgment $t_1 \longleftrightarrow t_2$ is used in rewrite derivations. Derived rewrites are declared with the **interactive_rw** form, as follows:

interactive_rw *name* : $t_1 \longleftrightarrow t_2$

When a rewrite is declared, the MetaPRL refiner requires a proof of the judgment $t_1 \longleftrightarrow t_2$. The judgment is not conditional, and it is not stated in a sequent calculus.

The Base_rewrite module lifts the rewrite judgment to the sequent level. It also defines rules for reflexivity and symmetry.

12.10.1 Parents

include Base_auto_tactic

12.10.2 Rules

The following rule defines the rewrite reflexivity. A term a always rewrites to itself. The judgment is stated using the built-in primitive rewrite judgment.

! **rule** rewriteAxiom1 H:
[*ext*] $H \vdash a \longleftrightarrow a$
= Base_rewrite!rw_just

The `rewriteAxiom2` conditional rewrite provides a link to the primitive rewriter: a proof of $a \longleftrightarrow b$ shows that the terms a and b are computationally equivalent.

![] **rewrite** `rewriteAxiom2` $ab : a \longleftrightarrow b \rightarrow a \longleftrightarrow b$

A rule for symmetry is also defined. The rules for symmetry and transitivity and symmetry can be derived from reflexivity `rewriteAxiom1` and substitution `rewriteAxiom2`.

\# **rule** `rewriteSym` H:
[*ext*] $H \vdash a \longleftrightarrow b \rightarrow$
[*ext*] $H \vdash b \longleftrightarrow a$

The reflexive rule `rewriteAxiom1` is also added to the `trivialT` resource.

Chapter 13

First-Order Logic

In this tutorial we develop an account of intuitionistic and classical first-order logic, and we relate it to the Nuprl type theory `Itt_theory`. We present the account as a tutorial in MetaPRL. The theory we develop is fully modular. It contains modules for each of the logical operators, and the constructive logic is a subtheory of the classical logic. We also investigate proof automation. Each module defines proof procedures for its operator, and the final logic defines an automated proof procedure for the logic.

The first step in creating a logic is creating a module for each of the logical operators. Initially, the framework has no notion of truth or falsehood in the logic; these concepts are defined by declaring syntax and logical rules. Our logic will include the logical operators in first-order logic (`FOL`), and it will include terms to represent *true* and *false*. We will use a single-conclusion sequent calculus (a multi-conclusion sequent calculus is also possible, but this choice will make it easier to relate the logic to the type theory).

This presentation is divided into several parts. In Section 13.1, we develop a primitive account of the constructive part of first-order logic; then in Section 13.2 we define an interpretation in the Nuprl type theory. The constructive logic is extended to a classical first-order logic in Section 13.3, but the interpretation in Section 13.4 in the Nuprl type theory must be significantly modified.

13.1 Defining a First-Order Logic

The presentation takes the form of a tutorial, and we will explicitly go through all the steps necessary define the First-Order logic and relate it to the type theory.

Table 13.1: Listing of the `theories` directory

```
<tulare: 14>cd ~/meta-prl/theories
<tulare:theories 15>ls
CVS/  czf/  lf/        reflect_itt/  tactic/
base/ fol/  ocaml/     rewrite/      tptp/
caml/ itt/  ocaml_sos/ sil/          tutorial/
```

13.1.1 Getting started

Before we begin, we need to configure the MetaPRL build-system to recognize the new theory we are creating. Normally, logical theories are placed in separate directories within the subdirectory *MetaPRL-main-directory*/`theories`. Table 13.1 shows a listing of the standard distribution. Each of the theory directories contains a definition of a logic, or support for a logic.

base base logic, including basic display forms for modules, and base resources for proof automation,

caml a definition of Caml programs,

czf Aczel's CZF set theory,

fol a complete definition of first-order logic,

itt the Nuprl type theory,

lf the Edinburgh Logical Framework (incomplete),

ocaml definitions and display forms for OCaml programs,

ocaml_sos an operational semantics for OCaml programs,

reflect_itt a theory of reflection for the Nuprl type theory,

rewrite an incomplete theory for testing the term rewriter,

sil the definition and semantics of a "simple imperative language,"

tactic basic definitions of tactics and tacticals,

tptp an interface to the TPTP set of problems for theorem provers,

tutorial the directory for this tutorial

We will use the tutorial directory for defining the logic in this tutorial. To add this directory to the build system, the file mk/config must be modified to include the tutorial directory in the build. This is specified by the THEORIES variable, which should be modified to include the tutorial directory as follows.

```
THEORIES=tactic ocaml base itt tutorial
```

The tactic, ocaml, and base theories are always included in the build; we include the itt theory in the build because we will be relating our first-order logic to the Nuprl type theory, and we also include the tutorial directory.

MetaPRL uses the Unix make command to build the system, so we will need to create a Makefile in the tutorial directory. This file is boilerplate, and all theories have the basic Makefile contents shown in Table 13.2. Most of the actual make definitions are in the $(ROOT)/mk/rules file; the Makefile contains only the necessary information for the theory, including the name of the theory (DIR), the location of the MetaPRL root directory (ROOT), a list of directories to search for theory files (INCLUDE), and a list of module in this theory (MPFILES). As files are added to the theory, they must also be added to the MPFILES variable to be included in the build (at the moment there are no files in the theory yet).

Once the mk/config file has been edited, and the theories/tutorial/Makefile created, MetaPRL can be built by using the make command, as shown in Table 13.3.

13.1.2 Defining well-formed formulas

The first step in creating the logic is to declare a term to define *well-formedness* of expressions. We use the term type to provide a well-formedness type judgment. Proofs of well-formedness are computationally meaningless, and we also declare a trivial proof term trivial. To form this judgment we create a module Fol_type. First, an *interface* needs to be defined that declares the syntax of this type judgment.

• *Create the file* fol_type.mli *containing the lines:*

```
declare "type"{'A}
declare trivial
```

The intent of the "type" term is that if the expression type{'t} is provable, then the term 't is a well-formed formula.

Next, we create the implementation. The rules for well-formedness will be stated separately in the modules that define the logical operators,

Table 13.2: Boilerplate `Makefile` code

```
DIR := theories/tutorial
ROOT := ../..
INCLUDE := -I . -I ../itt -I ../tactic\
   -I ../ocaml -I ../base -I $(ROOT)/lib
OCAMLDEPFLAGS := -prl

# Theory files
MPFILES :=

# This part is standard
MAIN = theory
all: $(MAIN).cma
opt: $(MAIN).cmxa

clean::
        $(RM) *.p4* *.pp*

# Real makefile
include $(ROOT)/mk/rules
include Makefile.dep
```

Table 13.3: Building MetaPRL

```
<tulare 182>cd ~/meta-prl
<tulare 183>make
Making util...
Making clib...
Making mllib...
Making refiner...
....more directories....
Making theories/tactic...
Making theories/ocaml...
Making theories/base...
Making theories/itt...
Making theories/tutorial...
Making editor/ml...
```

so the only implementation we need to include is the syntax declaration and a display form to produce pretty output.

• *Create the file* `fol_type.ml`, *containing the following lines:*

```
include Base_theory
declare "type"{'A}
declare trivial
dform type_df : "type"{'A} = slot{'A} `" type"
dform trivial_df : trivial = cdot
```

The `Base_theory` module is included for basic logic and display form definitions. The display forms define the print representations of the two terms we have just defined. The term `type{'t}` is displayed character ·, which is defined in the `Nuprl_font` module in the `Base_theory` theory.

The next step is to add the new module to the build system.

• *Edit the* `Makefile`, *and add the following line:*

```
MPFILES := fol_type
```

• *Run the* `make` *command*

```
% make
...compilation messages...
```

This step creates several new files.

fol_type.cmi is the compiled interface file,

fol_type.cmo is the compiled implementation file,

fol_type.ppo is a version of the implementation file that has been preprocessed by the MetaPRL compiler,

fol_type.cmiz summarizes the logical content of the interface file `fol_type.mli`,

fol_type.cmoz summarizes the logical content of the implementation file `fol_type.ml`.

The implementation is compared with the interface during the compilation, and any discrepencies in the definitions will produce an error.

Now that the module is defined, we can view it with the module editor. The MetaPRL system must be built first, and this can be performed by running `make` in the MetaPRL root directory.

• *Compile the new editor, with the following commands:*

Table 13.4: Editor listing for the `fol_type` module

```
% cd editor/ml
% ./mptop
MetaPRL 0.6.0:
    build [Fri Sep 3 10:56:10 EDT 1999]
    on tulare.cs.cornell.edu
    Uses VERBOSE Refiner_ds
# cd "fol_type";;
/fol_type : string
# ls "";;
Implementation:
begin
    include Base_theory
    declare Fol_type!type{'t}
    declare Fol_type!trivial
    dform type_df : Fol_type!type{'t} = t" type"
    dform trivial_df : Fol_type!trivial = ·
    Id: 849069226
end
```

```
% cd ~/meta-prl
% make
...compilation messages...
```

• *Start the editor, and view the* `Fol_type` *module:*
The theory listing is shown in Table 13.4. It includes the declarations with their unqualified names (the full names are qualified by the module name as `Fol_type!type` and `Fol_type!trivial`). It also lists the display forms in expanded form. The editor can be exited with the command `exit 0;;`, or with EOF (usually `^D` in Emacs).

13.1.3 Logical constants

The next step will be to add modules for the logical constants *true* and *false*. Our logic will eventually have seven modules, one for each of the logical operators. Each module has an interface defined in a `.mli` file, and an implementation defined in a `.ml` file. The interface defines the syntax, and the implementation provides the rules, display, and tactics that implement the module.

The `False` interface

First, define the interface for the false operator in the module `Fol_false`.

• *Create the file* `fol_false.mli`*, containing the line:*

```
declare "false"
```

The declaration defines a term `"false"` to represent the constant *false*.

The `False` implementation

The implementation for the `false` constant contains the same syntax declaration as in the interface, it includes a display form, and it defines the basic rules for falsehood.

• *Create the file* `fol_false.ml`*, containing the following lines:*

```
include Fol_type

declare "false"
dform false_df : "false" = ‘"false"
prim false_type 'H :
 sequent ['ext] { 'H >- "type"{."false"} } = trivial

prim false_elim 'H 'J :
 sequent ['ext] { 'H; x: "false"; 'J['x] >- 'C['x] } =
 trivial
```

The `include` directive establishes the dependency of the `Fol_false` module on the `Fol_type` module, which is used in the well-formedness rule `false_type`. The display form for `false` is the string literal `"false."` The first inference rule, `false_type`, declares that `false` is a well-formed formula, and the second rule, `false_elim`, states that anything can be derived from a `false` assumption.

The rules are declared as primitive, since they are *axioms* in the first-order logic. There is no computational content to either rule, and the proof terms are the trivial proof `trivial`.

Next, we must add the `fol_false` module to the build system.

• *Modify the definition of* `MPFILES` *in the* `Makefile`*:*

```
MPFILES := fol_type fol_false
```

• *Build the new logic*

```
% make
...compile messages...
```

The `True` constant

The module defining the constant `true` is similar to the `false` module. The `Fol_true` module declares a new term for `true`, and it also defines the computational content of `true` as the proof term `it`. The contents of the `fol_true` module are shown below.

Interface:
```
declare "true"
declare it
```

Implementation:
```
include Fol_type

declare "true"
dform true_df : "true" = ‘"true"

prim true_type 'H :
   sequent ['ext] { 'H >- "type"{."true"} } = trivial

prim true_intro 'H :
   sequent ['ext] { 'H >- "true" } = it
```

13.1.4 Basic Automation

When a rule is declared, the MetaPRL compiler produces a *tactic* with the same name that can be used to apply the rule during a proof. For instance, the `false_type` rule has the following declaration:

```
# false_type;;
- : Refiner.Refiner.TermAddr.address ->
Tactic_type.Tactic.tactic = <fun>
```

A *tactic* is a function that applies the rule by backward-chaining and matching. For the `false_type` rule, the second-order variable `'H` represents an arbitrary list of hypotheses, and the tactic for `false_type` requires an explicit *address* argument that specifies how many hypotheses there are in the actual goal sequent.

The use of raw rules can be rather unwieldy. In each step of a proof, a particular rule must be selected for application, and its parameters have to be constructed explicitly. If a logic has more than a few rules, this process can be extremely time-consuming. The obvious solution is to provide proof search procedures that 1) automate the proof search process, and 2) categorize the *styles* of reasoning.

A simple example is helpful to describe this approach. In sequent calculi, rules are often classified as "introduction" and "elimination" rules. In a goal-directed proof, introduction rules decompose logical operators to the right of the sequent turnstile, and elimination rules decompose logical operators on the left. As we develop our first-order logic, we will need to provide elimination and introduction rules for each of the operators that we add—for conjunction, disjunction, implication, etc. Then, if we were to perform a proof, say of the sequent $A \Rightarrow B \Rightarrow C \vdash (A \wedge B) \Rightarrow C$, we would use a sequence of introduction and elimination rules.

$A \Rightarrow B \Rightarrow C \vdash (A \wedge B) \Rightarrow C$
BY implies_intro 1
$A \Rightarrow B \Rightarrow C,\ A \wedge B \vdash C$
BY and_elim 2
$A \Rightarrow B \Rightarrow C,\ A,\ B \vdash C$
...more...

This process is tiresome because in each step we have to remember the name of the rule to apply, and figure out its arguments. But intuitively, the sequence of steps is just a sequence of "decompositions" of logical operators. It would be much easier to have a *single* tactic that is able to perform general decompositions given the address of the term to decompose. As we define elimination and introduction rules, we would then add their descriptions to this generic tactic. A tactic like this is *context-sensitive:* its behavior depends on the set of rules that are within scope when the tactic is applied.

MetaPRL implements a general mechanism, called a *resource*, for defining context-sensitive tactics (and other kinds of functions). A resource has a name, and it is declared with four types:

resource (*insert_type*, *result_type*, *data_type*, *arg_types*) *name*.

A resource can be thought of abstractly as a collection of values of *insert_type*. A resource has two basic operations: *insertion* adds a new component of type *insert_type* to a resource, and *extraction* produces a value of type *result_type* from the resource. The *data_type* is the internal representation that the resource uses to represent the collection; we will postpose discussion of *arg_types* for the moment. The main difference between a resource and a more normal collection like a list is that MetaPRL maintains scoping behavior for resources. The value of a resource in a module includes all insertions from `included` modules, and all insertions in the current module.

Since sequent calculi are so common, the MetaPRL base theory defines resources for collecting introduction and elimination rules, and

using them to build a generic "decomposition" tactic dT. The definition of these resources are as follows:

```
resource (term * (int -> tactic),
int -> tactic, elim_data, elim_args) elim_resource
resource (term * tactic, tactic,
intro_data, intro_args) intro_resource.
```

Insertion into these resources requires a term argument that defines a pattern that is being decomposed as well as a tactic that performs the decomposition. The elimination resource also requires an integer argument the identifies the hypothesis that is being decomposed.

The dT tactic consults these resources when "decomposing" logical operators. It is declared as follows:

```
val dT : int -> tactic
```

The argument to dT is the clause number that the tactic is applied to: if the argument is 0, the tactic applies the introduction rule to the conclusion (dT is defined only for single-conclusion sequent calculi). Otherwise, the clause number specifies the hypthesis number for application of an elimination rule.

Manual addition to the dT tactic

We can add the introduction and elimination rules to the dT tactic by adding proof procedures to the intro_resource and elim_resource. To illustrate the process, we will add proof automation to the fol_false module. The first step is to add reasoning for the well-formedness of the false term.

• *Add an introduction tactic for proving well-formedness to the* Fol_false *module, by adding the following function to the* fol_false.ml *file after the* false_type *rule:*

```
let d_false_type p =
  false_type (Sequent.hyp_count_addr p) p
```

The d_false_type function is a *tactic*. As defined in the module TacticType, a tactic is a function of type tactic_arg -> tactic_value, where tactic_value is an abstract type, and tactic_arg is a data type containing the goal term that is to be refined. The false_type rule requires an *address* argument that defines the number of hypotheses in the goal sequent (the 'H argument); the Sequent module defines some common functions for constructing

addresses from tactic arguments. The false_type rule has type address -> tactic_arg -> tactic_value, and so the function d_false_type has type tactic_arg -> tactic_value = tactic. The d_false_type function is just a representative of the false_type rule, where the address is computed automatically.

We can now add this rule to the intro_resource.

• *Add the introduction rule to the* intro_resource *by adding the following line after* d_false_type *function:*

```
let intro_resource =
 Mp_resource.improve intro_resource
  (<< "type"{."false"} >>, d_false_type)
```

This definition adds the d_false_type tactic to the intro_resource to be used when decomposing a term of the form "type"{."false"}. The update is functional; it returns a new copy of the resource where the insertion has been performed.

The same sequence of steps can be used to add the false_elim rule to the elim_resource.

• *Define a tactic for the* false_elim *rule:*

```
let d_false_elim i p =
 let j, k = Sequent.hyp_indices p i in
  false_elim j k p
```

This tactic has the same pattern as the tactic for false_type, but it requires an iteger argument i that specifies the hypothesis to decompose. The Sequent module provides a function that converts a hypothesis number into two addresses that identify the sequence of hypotheses before and after the specified hypothesis. The resulting type of d_false_elim is int -> tactic.

The final step is to add this to the elimination resource.

• *Add the* d_false_elim *tactic to the* elim_resource *by adding the following lines after the* d_false_elim *function:*

```
let elim_resource =
  Mp_resource.improve elim_resource
   (<< "false" >>, d_false_elim)
```

At this point, the automation is complete (at least for adding the rules for false to the dT tactic), but the proofs we can perform with just the term false are rather limited. We will revisit proof automation after we have added more operators to the theory. In the meantime, let's return to a discussion of the fourth component to a resource, of type *args_type*.

Automatic addition of automation to `Fol_true`

If we continue adding automation to the `fol_true` module, we will notice that the steps are almost identical to adding automation to the `fol_false` module: first we would define tactics that provide the address arguments to the rules, and then we would add the tactics to the `intro_resource` resource. These steps are all boilerplate code, and adding the proof automation manually results in much the same code in almost all situations.

To address this problem, MetaPRL has an alternative method for automatically adding rules to resources as the rules are defined. The fourth argument of a resource specifies an *annotation* that can be attached to a rule to add it directly to the resource. For the `intro_resource` and `elim_resource`, these extra arguments are just a list of options:

```
type elim_args = pre_tactic * elim_option list
type intro_args = pre_tactic * intro_option list
```

The type `pre_tactic` is a rule description automatically provided by MetaPRL when the resource is added; we won't need to know anything more about it. The options are also for more esoteric uses of the resources, and for this tutorial we won't need to provide any options.

Automatic insertion into resources is performed by adding *annotations* to the rule definitions. The annotation is enclosed in $\{|\cdots|\}$ brackets just after the *name* of the rule and before any extra argument. The annotation includes the name of a resource and any additional arguments.

• *Add resource annotations to the rules in the* `Fol_true` *module, by modifying the rule definitions as follows:*

```
prim true_type {| intro_resource [] |} 'H :
   sequent ['ext] { 'H >- "type"{."true"} } = trivial
prim true_intro {| intro_resource [] |} 'H :
   sequent ['ext] { 'H >- "true" } = it
```

These rule annotations add the rules directly to the `intro_resource`. Essentially, the MetaPRL compiler constructs a version of the code that we added manually for the `Fol_false` module, automatically figuring out default values for the arguments of the rule. The `[ ]` brackets are the empty list of options passed to the `intro_resource`; we don't need to pass any extra arguments, and this list is empty.

13.1.5 Adding the propositional connectives

The next step in the development of the first-order logic is to define the propositional connectives for conjuction, disjunction, implication, and negation. In this section, we will show how to add the definition of implication; the modules for conjuction and disjunction are similar. Negation can be derived from implication and the constant false, as we show in the next section.

The interface for implication follows the standard format. We include the `Fol_type` module to define well-formedness of the term, and we declare a new binary operator for `implies`. We also define proof terms `lambda` and `apply`, and display form precedences for each of the new terms.

• *Define the* `Fol_implies` *interface by adding the following lines to the file* `fol_implies.mli`*:*

```
include Fol_type

prec prec_implies
prec prec_lambda
prec prec_apply

declare implies{'A; 'B}
declare lambda{x. 'b['x]}
declare apply{'f; 'a}
```

The implementation for the `implies` term includes the declarations of the interface, and it also defines a display form, and the introduction and elimination rules for implication.

• *Declare the syntax for the* `implies` *operator by adding the following lines to the file* `fol_implies.ml`*:*

```
include Fol_type

declare implies{'A; 'B}
declare lambda{x. 'b['x]}
declare apply{'f; 'a}
```

The next step is to provide display forms that print these terms in their more standard representations. The display forms for these terms require parenthesization for non-ambiguous parsing, so we introduce several precedence declarations.

• *Provide precedence declarations for pretty-printing, by adding the following lines to the* `fol_implies.ml` *file:*

```
prec prec_implies
prec prec_lambda
prec prec_apply

prec prec_lambda < prec_apply
prec prec_lambda < prec_implies
prec prec_implies < prec_apply
```

These precedence declarations require a partial order on the display precedences

prec_lambda < prec_implies < prec_apply.

The display forms are added with dform declarations.

- *Add display forms for each of the terms in* Fol_implies*:*

```
dform implies_df : parens ::
   "prec"["prec_implies"] :: implies{'A; 'B} =
   szone pushm[0] slot["le"]{'A} hspace
      Rightarrow `" " slot{'B} popm ezone
dform lambda_df : parens ::
   "prec"["prec_lambda"] :: lambda{x. 'b} =
   szone pushm[3] Nuprl_font!lambda slot{'x} `"."
      slot{'b} popm ezone
dform apply_df : parens ::
   "prec"["prec_apply"] :: apply{'f; 'a} =
   slot{'f} hspace slot{'a}
```

There are several parts to these display form definitions. Display forms contain *zones* that determine the behavior of line-breaking within the zone. A *soft* zone (defined with the term szone) has the behavior that either 1) *all* hard-breaks are taken (the term hspace is a hard-break that either prints a space character or causes a line break), or 2) *none* of them are. The display forms also contain indentation zones, defined with the terms pushm and popm. A pushm[i] term indents all lines up until the next popm term to the current column plus i characters.

The slot terms are used to display subterms and perform parenthesization. In the implies_df display form, the slot{'A} term displays the subterm represented by 'A, wrapping the displayed representation of 'A in parentheses if the precedence of the display form for 'A is *less* than prec_implies. The term slot["le"]{'A} displays 'A with parentheses if its precedence is *no more than* prec_implies.

The Rightarrow term is defined in the module Nuprl_font, and it displays as the character ⇒. These definitions result in the following kinds of output.

Term	Display
`implies{true; true}`	$True \Rightarrow True$
`implies{implies{'A; 'B}; 'C}`	$(A \Rightarrow B) \Rightarrow C$
`implies{'A; implies{'B; 'C}}`	$A \Rightarrow B \Rightarrow C$
`implies{'A; apply{'B; 'C}}`	$A \Rightarrow (B\ C)$

The `Fol_implies` module defines computation over the proof terms `lambda` and `apply`. Computation uses the standard beta-reduction form.

• *Define computation over proof terms with the following rewrite rule:*

```
prim_rw beta : apply{lambda{x. 'b['x]}; 'a} <--> 'b['a]
```

This rewrite states that the beta-redex $(\lambda x.b[x])(a)$ is computationally *equivalent* to the term $b[a]$, where a has been substituted for x in $b[x]$.

The implication has introduction and elimination rules corresponding to natural deduction rules for implication: in order to prove $(A \Rightarrow B)$, assume A and prove B; if A is provable with assumption $(A \Rightarrow B)$, then B is also a valid assumption. We add the inference rules as follows.

• *Add the inference rules for implication:*

```
prim implies_type {| intro_resource [] |} 'H :
   sequent ['ext] { 'H >- "type"{'A} } -->
   sequent ['ext] { 'H >- "type"{'B} } -->
   sequent ['ext] { 'H >- "type"{implies{'A; 'B}} }
   = trivial
prim implies_intro {| intro_resource [] |} 'H 'x :
   sequent ['ext] { 'H >- "type"{'A} } -->
   ('b['x] : sequent ['ext] { 'H; x: 'A >- 'B }) -->
   sequent ['ext] { 'H >- 'A => 'B }
   = lambda{x. 'b['x]}
prim implies_elim {| elim_resource [] |} 'H 'J 'b :
   ('a : sequent ['ext]
      { 'H; f: 'A => 'B; 'J['f] >- 'A }) -->
   ('x['b] : sequent ['ext]
      { 'H; f: 'A => 'B; 'J['f]; b: 'B >- 'C['f] }) -->
   sequent ['ext]
      { 'H; f: 'A => 'B; 'J['f] >- 'C['f] } = 'x['f 'a]
```

The proof terms for the rules use second-order binding. For instance, the proof term for the `implies_intro` rule states that if the computational content of the second clause is `'b['x]`—that it is a term with a free variable x—then the computational content of the goal clause is $\lambda(x.b[x])$.

The modules for conjuction and disjunction are similar to implication. These modules are shown in tables 13.5, 13.6, and 13.7. The module for conjuction represents the computational content of the conjuction $A \wedge B$ as the pair (a, b) where a is the content of the proposition A, and b is the content of the proposition B. The `spread` term is the `pair` destructor. In this module we have also added *label* annotations to the rules with prefixes of the form `[wf]` and `[main]`. These labels are assigned to the subgoals during refinement, and are displayed as labels on the proofs.

The module for disjunction is similar: the computational content of a disjunction $A \vee B$ is either the term $inl(a)$ or $inr(b)$ where a is the computational content of A, and b is the computational content of B. The `decide` term is the corresponding destructor. The disjunction has *two* introduction rules, one for each branch of the disjunction. This is a special case where the `intro_resource` requires an argument to distinguish selection of the rule to apply during disjunction decomposition. The option `SelectOption` is defined in the `Base_dtactic` module, and it means that the `dT` tactic requires a "selection" argument (either 1 or 2 in this case) that identifies the rule to apply during decomposition. The correct usage of the `dT` tactic for disjunction-introduction requires the use of the `selT : int -> tactic -> tactic` *tactical* to provide the selection argument. Valid uses are: `selT 1 (dT 0)` and `selT 2 (dT 0)`.

13.1.6 Defining negation

Negation is a special case of propositional operator because it can be defined in terms of implication and falsehood, with the definition $\neg A \equiv A \Rightarrow \mathit{False}$. It is possible to define the implication and introduction rules for negation directly, but it is better to derive the rules from the rules for negation and falsehood, rather than defining them as primitive.

The interface for negation declares the syntax as usual, but it establishes a dependency on the `Fol_false` and `Fol_implies` modules, and it declares the negation definition.

• *Declare the interface for negation by adding these lines to the file* `fol_not.mli`*:*

```
include Fol_false
include Fol_implies
declare "not"{'A}
rewrite unfold_not : "not"{'A} <-->
   implies{'A; ."false"}
prec prec_not
```

Table 13.5: Fol_and module for conjuction

```
include Fol_type

declare "and"{'A; 'B}
declare "pair"{'a; 'b}
declare spread{'x; a, b. 'body['a; 'b]}

prec prec_and

dform and_df : parens :: "prec"["prec_and"] ::
   "and"{'A; 'B} =
   szone pushm[0] slot{'A} hspace wedge
   `" " slot{'B} popm ezone

dform pair_df : "pair"{'a; 'b} =
   `"<" slot{'a} `"," slot{'b} `">"

dform spread_df : "spread"{'x; a, b. 'body} =
   szone pushm[0] `"let <" slot{'a} `"," slot{'b}
   `"> =" hspace slot{'x} hspace `"in" hspace
   slot{'body} popm ezone

prim_rw reduce_spread :
   spread{pair{'x; 'y}; a, b. 'body['a; 'b]} <-->
     'body['x; 'y]

prim and_type {| intro_resource [] |} 'H :
   [wf] sequent ['ext] { 'H >- "type"{'A} } -->
   [wf] sequent ['ext] { 'H >- "type"{'B} } -->
   sequent ['ext] { 'H >- "type"{.'A & 'B} } = trivial

prim and_intro {| intro_resource [] |} 'H :
   [main] ('a : sequent ['ext] { 'H >- 'A }) -->
   [main] ('b : sequent ['ext] { 'H >- 'B }) -->
   sequent ['ext] { 'H >- 'A & 'B } = pair{'a; 'b}

prim and_elim {| elim_resource [] |} 'H 'J 'x 'y 'z :
   [main] ('body['y; 'z] : sequent ['ext]
    { 'H; y: 'A; z: 'B; 'J['y, 'z] >- 'C['y, 'z] }) -->
   sequent ['ext] { 'H; x: 'A & 'B; 'J['x] >- 'C['x] }
   = spread{'x; y, z. 'body['y; 'z]}
```

Table 13.6: Fol_or module for disjunction (part 1)

```
include Fol_type
open Base_dtactic

declare "or"{'A; 'B}
declare inl{'a}
declare inr{'b}
declare decide{'x; y. 'body1['y]; z. 'body2['z]}

prec prec_or
prec prec_inl
prec prec_inr
prec prec_decide

dform or_df : parens :: "prec"["prec_or"] ::
   "or"{'A; 'B} =
   szone pushm[0] slot{'A} hspace
   vee `" " slot{'B} popm ezone

dform inl_df : parens :: "prec"["prec_inl"] ::
   inl{'x} = `"inl " slot{'x}

dform inr_df : parens :: "prec"["prec_inl"] ::
   inr{'x} = `"inr " slot{'x}

dform decide_df : parens :: "prec"["prec_decide"] ::
   decide{'x; y. 'body1; z. 'body2} =
   szone pushm[3] `"match " slot{'x} `" with" hspace
   `"inl " slot{'y} `" ->" hspace slot{'body1}
   hspace `"| inr " slot{'z} `" ->" hspace slot{'body2}
   popm ezone

prim_rw reduce_decide_inl :
   decide{inl{'x}; y. 'body1['y]; z. 'body2['z]} <-->
   'body1['x]
prim_rw reduce_decide_inr :
   decide{inr{'x}; y. 'body1['y]; z. 'body2['z]} <-->
   'body2['x]
```

Table 13.7: Fol_or module for disjunction (part 2)

```
(* Rules *)
prim or_type {| intro_resource [] |} 'H :
   [wf] sequent ['ext] { 'H >- "type"{'A} } -->
   [wf] sequent ['ext] { 'H >- "type"{'B} } -->
   sequent ['ext] { 'H >- "type"{."or"{'A; 'B}} } =
   trivial

prim or_intro_left
   {| intro_resource [SelectOption 1] |} 'H :
   [wf] sequent ['ext] { 'H >- "type"{'B} } -->
   [main] ('a : sequent ['ext] { 'H >- 'A }) -->
   sequent ['ext] { 'H >- "or"{'A; 'B} } =
   inl{'a}

prim or_intro_right
   {| intro_resource [SelectOption 2] |} 'H :
   [wf] sequent ['ext] { 'H >- "type"{'A} } -->
   [main] ('b : sequent ['ext] { 'H >- 'B } ) -->
   sequent ['ext] { 'H >- "or"{'A; 'B} } =
   inr{'b}

prim or_elim {| elim_resource [] |} 'H 'J 'x :
   [wf] ('a['x] : sequent ['ext]
      { 'H; x: 'A; 'J[inl{'x}] >- 'C[inl{'x}] }) -->
   [wf] ('b['x] : sequent ['ext]
      { 'H; x: 'B; 'J[inr{'x}] >- 'C[inr{'x}] }) -->
   sequent ['ext]
      { 'H; x: 'A or 'B; 'J['x] >- 'C['x] } =
   decide{'x; x. 'a['x]; x. 'b['x]}
```

The include directives establish the logical module dependencies. The declare defines the syntax for negation, and the rewrite term establishes the definition of not as a computational equivalence. Computational rewriting is the general definitional mechanism in MetaPRL: this definition states that, in all contexts, the term not{'A} is equivalent to the term implies{'A; false}.

In the implementation file, the syntax for negation is established just as in the interface.

• *Define the logical dependencies and syntax for negation in the* fol_not.ml *file:*

```
include Fol_false
include Fol_implies
declare "not"{'A}
```

The display form for negation has the standard format. We include a precedence declaration for parenthesization.

• *Define the display properties of negation:*

```
prec prec_not
dform not_df : parens :: "prec"["prec_not"] ::
   "not"{'A} = tneg slot{'A}
```

Negation is defined as in the interface, as an implication of falsehood.

• *Provide the definition of negation as implication of falsehood:*

```
prim_rw unfold_not : "not"{'A} <-->
   implies{'A; ."false"}
```

Note that this declaration in the implementation states that the definition is primitive, where the interface simply declared the rewrite with the rewrite directive.

The rules for negation include the standard properties of well-formedness and introduction and elimination rules, but these rules can be derived from the rules for false and implies. In the fol_not.ml file, we declare the rules as *interactive*, which states that the rules are presumed valid with a proof obligation to show that they can be derived.

• *Define the derived inference rules for negation:*

```
interactive not_type {| intro_resource [] |} 'H :
 sequent ['ext] { 'H >- "type"{'A} } -->
 sequent ['ext] { 'H >- "type"{."not"{'A}} }
interactive not_intro {| intro_resource [] |} 'H 'x :
 sequent ['ext] { 'H; x: 'A >- "false" } -->
 sequent ['ext] { 'H; x: 'A >- "not"{'A} }
interactive not_elim {| elim_resource [] |} 'H 'J :
 sequent ['ext] { 'H; x: "not"{'A}; 'J['x] >- 'A } -->
 sequent ['ext] { 'H; x: "not"{'A}; 'J['x] >- 'C['x] }
```

The next step is to *prove* these inference rules, which requires that we use the interactive proof editor. During this proof, we will be making use of the `dT` tactic to apply the introduction and elimination rules for `false` and `implies`. To start the proof, we need to compile a build of the system. The `MPFILES` variable in the `Makefile` should contain a list of all the modules in the tutorial so far:

```
MPFILES := fol_type fol_false fol_true\
   fol_implies fol_and fol_or fol_not
```

• *Build a version of MetaPRL that contains all the modules in the tutorial:*

```
~/meta-prl/tutorial% cd ..
~/meta-prl% make
...messages from compiler...
```

• *Start the proof editor, and list the* `Fol_not` *module (Table 13.8, Step 1).*

This listing, shown in Table 13.8, shows that there are three incomplete proof obligations: the rules for `not_type`, `not_intro`, and `not_elim`. We will walk through a proof of these rules.

• *Change the current directory to the* `not_type` *rule (Table 13.9, Step 2).*

To prove this goal, we first unfold the negation using the definition `unfold_not`. The `unfold_not` definition is a *computational rewrite*, also called a *conversion*. Rewrites are like tactics: there are *conversionals* to combine rewrites and perform search; these conversionals are defined in the module `Tactic_type.Conversionals`. for our purposes in applying the `unfold_not` rewrite, we can use the `rwh` ("rewrite higher") that searches for the outermost occurrences of terms to rewrite. The `rwh` function is declared as `rwh : conv -> int -> tactic`: it takes a `conv` (a rewrite), and a clause number, and produces a `tactic` that applies the rewrite to that clause, rewriting the outermost terms.

• *Unfold the definition of the negation (Table 13.9, Step 3).*

The unfolding produced one subgoal where the definition of negation has been unfolded. We can descend into the subgoal using the

Table 13.8: Listing of the `Fol_not` module

Step 1

```
~/meta-prl% cd editor/ml
~/meta-prl/editor/ml% ./mptop
# cd "/fol_not";;
  /fol_not : string
# ls "";;
```

Implementation:
begin
  include Fol_false
  include Fol_implies
  declare Fol_not!not{'A}
  prec prec_not
  dform not_df **: prec[**prec_not**] :: parens ::** Fol_not!not{'A} = $\neg A$
  ! **rewrite** unfold_not : $(\neg A) \longleftrightarrow (A \rightarrow \text{False})$
  #[0,1] **rule** not_type H: [*ext*] $H \vdash A$ type $\longrightarrow$ [*ext*] $H \vdash (\neg A)$ type
  # **rule** not_intro H x:
    ["wf**"]** [*ext*] $H \vdash A$ type $\longrightarrow$
    ["main**"]** [*ext*] H; x: $A \vdash$ False $\longrightarrow$
    [*ext*] $H \vdash \neg A$
  # **rule** not_elim H J:
    ["main**"]** [*ext*] H; x: $\neg A$; $J[x] \vdash A \longrightarrow$
    [*ext*] H; x: $\neg A$; $J[x] \vdash C[x]$
  Id: 820346913
end

Table 13.9: Proving the `not_type` rule (steps 2, 3)

Step 2

```
# cd "not_type";;
  /fol_not/not_type : string
# ls "";;
  # [#]
  ....main....
  1. [ext] H⊢ A type
  ====
  [ext] H⊢ (¬A) type
  BY <goal>
```

Step 3

```
# refine rwh unfold_not 0;;
  # [*]
  ....main....
  1. [ext] H⊢ A type
  ====
  [ext] H⊢ (¬A) type
  BY rwh unfold_not 0
  1. [#]
     ....main....
     [ext] H⊢ (A → False) type
```

`down` proof-navigation function, then apply the `dT` tactic to apply the `implies_type` rule.

• *Navigate to the subgoal and apply the* `dT 0` *tactic (Table 13.10, Step 4).*

This proof step produces two subgoals: one to prove that `'A` is a type, and another to prove that `false` is a type. For the first goal, we use the typehood assumption, using that tactic `nthAssumT`.

• *Navigate to the first subgoal, and apply the* `nthAssumT 1` *tactic (Table 13.10, Step 5).*

Note that the application of the nthAssumT tactic produced a * on the status line, meaning that this branch of the proof is complete. To knock off the second subgoal, we navigate to it with the `up` and `down` functions, and we apply the `dT` tactic to invoke the `false_type` rule.

• *Navigate to the second subgoal, and apply the* `dT 0` *tactic (Table 13.11, Step 6).*

At this point, all proof obligations have been satisfied, and the proof is complete.

This was a pretty simple proof, involving only the use of the `dT` and `nthAssumT` tactics. The `autoT` tactic performs "automated" proving based on repeated application of several "basic" tactics including `dT` and `nthAssumT`. We can navigate back up the proof tree, and prove the goal in one step.

• *Navigate back to the root, and apply the* `autoT` *tactic (Table 13.11, Auto).*

The `autoT` tactic proves the goal, and it preserved the previous subtree (the root is listed under the `BY autoT` line). The infix function `thenT` is a *tactical* used for sequencing: the proof first unfolds the `not`

Table 13.10: Proving the `not_type` rule (steps 4, 5)

Step 4

```
# down 1;;
```

```
# # [#]
....main....
1. [ext] H⊢ A type
====
[ext] H⊢ (A → False) type
BY <goal>
```

```
# refine dT 0;;
```

```
# # [*]
....main....
1. [ext] H⊢ A type
====
[ext] H⊢ (A → False) type
BY dT 0
1. [#]
   ....wf....
   [ext] H⊢ A type
2. [#]
   ....wf....
   [ext] H⊢ False type
```

Step 5

```
# down 1;;
```

```
# # # [#]
....wf....
1. [ext] H⊢ A type
====
[ext] H⊢ A type
BY <goal>
```

```
# refine nthAssumT 1;;
```

```
# # * [*]
....wf....
1. [ext] H⊢ A type
====
[ext] H⊢ A type
BY nthAssumT 1
```

Table 13.11: Proving the `not_type` rule (step 6)

Step 6

```
# up 1;;
```

```
# # [*]
....main....
1. [ext] H⊢ A type
====
[ext] H⊢ (A → False) type
BY dT 0
1. [*]
   ....wf....
   [ext] H⊢ A type
2. [#]
   ....wf....
   [ext] H⊢ False type
() : unit
```

```
# down 2;;
```

```
# # # [#]
....wf....
1. [ext] H⊢ A type
====
[ext] H⊢ False type

BY <goal>

() : unit
```

```
# refine dT 0;;
```

```
* * * [*]
....wf....
1. [ext] H⊢ A type
====
[ext] H⊢ False type
BY dT 0
```

Auto

```
# cd "/fol_not/not_type";;
```

```
/fol_not/not_type : string
```

```
# refine rwh unfold_not 0
      thenT autoT;;
```

```
* [*]
....main....
1. [ext] H⊢ A type
====
[ext] H⊢ (¬A) type
BY rwh unfold_not 0 thenT autoT
====
1. *
   ....main....
   [ext] H⊢ (A → False) type
```

Table 13.12: Proof of the `not_intro` rule

```
# cd "/fol_not/not_intro";;
  /fol_not/not_intro : string
# refine rwh unfold_not 0 thenT autoT;;
  * [*]
  ....main....
  1. [ext] H⊢ A type
  2. [ext] H; x: A⊢ False
  ====
  [ext] H⊢ ¬A
  BY rwh unfold_not 0 thenT autoT
```

term, and *then* applies the `autoT` tactic.

The next proof that we must perform is for the rule `not_intro`. Like the `not_type` rule, the `not_intro` rule can be proved in one step.

• *Prove the* `not_intro` *rule in one step (Table 13.12).*

The `not_elim` rule is the final piece of reasoning that we have to perform. The `autoT` tactic does not completely prove this goal because it requires *folding* the definition of `not`. The root of the proof tree can be expanded using the `autoT` tactic, which leaves a partial proof.

• *Expand the initial part of the* `not_elim` *proof (Table 13.13, Step 1).*

The second goal has a trivial proof.

• *Prove the second subgoal by using the* `dT 4` *tactic (Table 13.14).*

The first goal is more difficult. The goal uses the hypothesis $A \Rightarrow$ *false*, and the assumption uses the hypothesis $\neg A$. MetaPRL allows rewriting on an assumption using the `rwch` rewriting function. The `rwch` function takes two numbers: the first number is the number of an assumption, and the second number is the hypothesis to rewrite.

• *Prove the first subgoal using the* `rwch` *rewriting function (Table 13.15).*

This completes the derivation. As a final step, we can change back to the module directory and check that all obligations have been satisfied. The `ls "u"` command lists only the unproved rules in a module. After listing, we can *save* the theory to the logical library.

• *Check that all obligations are satisfied and save the module (Table 13.16).*

Table 13.13: Prove the `not_elim` rule (step 1)

Step 1

```
# cd "../not_elim";;
```

```
/fol_not/not_elim : string
```

```
# ls "";;
```

```
# [#]
....main....
1. [ext] H; x: ¬A; J[x]⊢ A
====
[ext] H; x: ¬A; J[x]⊢ C[x]
BY <goal>
() : unit
```

```
# refine rwh unfold_not 2 thenT dT 2;;
```

```
# [*]
....main....
1. [ext] H; x: ¬A; J[x]⊢ A
====
[ext] H; x: ¬A; J[x]⊢ C[x]
BY rwh unfold_not 2 thenT dT 2
1. [#]
   ....assertion....
   [ext] H; x: A → False; J[x]⊢ A
2. [#]
   ....main....
   [ext] H; x: A → False; J[x]; b: False⊢ C[x]
```

Table 13.14: Prove the `not_elim` rule (step 2)

Step 2

```
# down 2;;
```

```
# # [#]
....main....
1. [ext] H; x: ¬A; J[x]⊢ A
====
[ext] H; x: A → False; J[x]; b: False⊢ C[x]
BY <goal>
() : unit
```

```
# refine dT 4;;
```

```
# * [*]
....main....
1. [ext] H; x: ¬A; J[x]⊢ A
====
[ext] H; x: A → False; J[x]; b: False⊢ C[x]
BY dT 4
```

Table 13.15: Prove the `not_elim` rule (step 3)

Step 3

```
# cd "../1";;
```

```
/fol_not/not_elim/1 : string
```

```
# refine rwch unfold_not 1 2 thenT nthAssumT 1;;
```

```
* * [*]
....assertion....
1. [ext] H; x: ¬A; J[x]⊢ A
====
[ext] H; x: A → False; J[x]⊢ A

BY rwch unfold_not 1 2 thenT nthAssumT 1
```

Table 13.16: Finish the `fol_not` module

```
# cd "~";;
  /fol_not : string
# ls "u";;
  Implementation:
  begin
  end
  () : unit
# save ();;
  () : unit
```

13.1.7 Adding structural rules

The logical primitives defined so far—`false`, `true`, `and`, `or`, `implies`, and `not`—together define a *constructive* propositional logic, but they are lacking some common *structural* rules. Structural rules define operations on *sequents* without an explicit connection to any particular logical connective. We need three structural rules: one for a proof by *assumption*, another for *thinning* of a hypothesis, and another for *cut* (addition of a lemma).

We will define these structural rules in a module called `Fol_struct`. The contents of the `fol_struct.ml` file is shown in Table 13.17.

The structural rules are defined in three parts: first the `include` and `open` statements establish the dependencies of the module. The `TacticType` module is opened for access to the `Sequent` module; the `TacticType.Tacticals` defines the `onSomeHypT` tactical used to augment the `trivialT` tactic, and the `Base_auto_tactic` defines the `trivialT` tactic.

The second parts lists the rules themselves. The `hypothesis` judgment specifies that if any term T is both an assumption and a goal, that the sequent is trivially provable. The `thin` judgment is a form of monotonicity: if a goal C is provable from the assumptions in H and J, then it remains provable with any additional assumption T. The `cut` rule is used to establish a lemma, and it can be thought of as a form of *modus-ponens*: if the lemma T can be proved, and the goal C can be established with the use of the lemma, then the goal C is provable from the original assumptions.

As with all the rules, rule application is rather unwieldy because of the necessity of providing the arguments (for addresses H and J and a

Table 13.17: Structural rules

```
include Base_theory

open Tactic_type
open Tactic_type.Tacticals

open Base_auto_tactic

(* Structural rules *)
prim hypothesis 'H 'J 'x :
   sequent ['ext] { 'H; x: 'T; 'J['x] >- 'T } = 'x

prim thin 'H 'J :
   ('t : sequent ['ext] { 'H; 'J >- 'C }) -->
   sequent ['ext] { 'H; x: 'T; 'J >- 'C } = 't

prim cut 'H 'T 'x :
   ('a : sequent ['ext] { 'H >- 'T }) -->
   ('b['x] : sequent ['ext] { 'H; x: 'T >- 'C }) -->
   sequent ['ext] { 'H >- 'C } = 'b['a]

(* Tactics *)
let nthHypT i p =
   let v, _ = Sequent.nth_hyp p i in
   let j, k = Sequent.hyp_indices p i in
      hypothesis j k v p

let thinT i p =
    let j, k = Sequent.hyp_indices p i in
        thin j k p

let assertT t p =
   let v = Var.maybe_new_vars1 p "v" in
      cut (Sequent.hyp_count_addr p) t v p

(* Add "hypothesis" to the trivialT tactic *)
let trivial_resource =
   Mp_resource.improve trivial_resource (**)
      { auto_name = "nthHypT";
        auto_prec = trivial_prec;
        auto_tac = onSomeHypT nthHypT
      }
```

new variable x in the `cut` rule), so the next step is to provide standard wrappers that compute the arguments. The `Sequent` module is used to compute the addresses, and the `Var` module is used to compute a new variable name for use in the `cut` rule.

The final step is to augment the standard tactics for reasoning. By convention, the `trivialT` tactic is a tactic (defined from the `trivial_resource`) that proves "trivial" goals. The definition of "trivial" is a matter of judgment, but it includes cases where the goal clause is also one of the assumptions, as in the `hypothesis` rule. The `trivial_resource` is defined as a component of the `autoT` tactic. The `autoT` tactic is essentially programmed with a list of tactics (in the `auto_resource` resource) that are each associated with a *precedence*. When the `autoT` tactic is applied to a goal, it applies each of the tactics in the list, from highest-to-lowest precedence until one of the tactics succeeds or all of them fail. If a tactic in the list succeeds and it produces subgoals, the `autoT` tactic repeats the process starting from the highest precedence tactic. If all tactics fail, the `autoT` tactic terminates.

To add a proof procedure to the `autoT` tactic, three values must be provided: a name (in `auto_name`) that is used for printing debugging information, and precedence (in `auto_prec`) that defines the precedence of the insertion, and the tactic itself (in `auto_tac`). The `Base_auto_tactic` module defines several precedences, and all additions to the `trivialT` tactic are at `trivial_prec`. The tactic we add is a version of the `nthHypT` tactic wrapped by the `onSomeHypT` tactical, which searches for the first hypothesis where the `nthHypT` tactic is successful.

13.1.8 Defining a propositional logic

The logical primitives defined so far—`false`, `true`, `and`, `or`, `implies`, and `not`—together define a complete *constructive* propositional logic. As the next step, we can define a module for propositional logic that puts all of the logical operators together. We will call this module `Fol_prop`.

• *Define a module for (constructive) propositional logic by creating two files* `fol_prop.mli` *and* `fol_prop.ml` *that both have the following contents:*

```
include Fol_type
include Fol_false
include Fol_true
include Fol_and
include Fol_or
include Fol_implies
include Fol_not
```

This module contains only `include` statements that state that the propositional logic is the combination of all the listed logics.

Let's prove a simple theorem in the logic that uses multiple connectives:

$$((A \vee B) \Rightarrow C) \Rightarrow ((A \Rightarrow C) \wedge (B \Rightarrow C)).$$

• *State the theorem by adding the following lines to the* `fol_prop.ml` *file:*

```
interactive distrib_or 'H :
   [wf] sequent ['ext] { 'H >- "type"{'A} } -->
   [wf] sequent ['ext] { 'H >- "type"{'B} } -->
   [wf] sequent ['ext] { 'H >- "type"{'C} } -->
   sequent ['ext] { 'H >- (('A or 'B) => 'C) =>
      (('A => 'C) & ('B => 'C)) }
```

The theorem has three assumptions stating that A, B, and C are *types*, which essentially means that they are assumed to be propositional letters. The theorem is stated within an *arbitrary* sequent context `'H`.

As mentioned previously, the `autoT` tactic is able to perform simple reasoning tasks, including repeated use of the `dT` tactic to decompose logical formulas into smaller and smaller parts. We can use `autoT` as a first step in the proof.

• *Use the* `autoT` *tactic to perform the initial steps of the proof (Table 13.18, Step 1).*

The `autoT` tactic has performed several steps at once. With enough experience, we can eventually figure out the expected behavior of the `autoT` tactic, but we can also *view* the steps that it took. To see what it did, we can navigate *into* the proof generated by `autoT` using the command `down 0`. There is a lot of detail and redundant information in these internal proofs, but we can eventually map the entire `autoT` subtree. The steps at each address are shown in Table 13.19.

Obviously, there is a lot of detail in the internal proof performed by `autoT`. However, the structure of the proof is apparent: the `autoT` tactic performs basic proof steps using the `dT 0` and `nthAssumT` tactics. The `dT` tactic in turn uses the introduction and elimination rules we

Table 13.18: Prove the `distrib_or` theorem

Step 1

[*]
....main....
1. [*ext*] $H \vdash A$ type
2. [*ext*] $H \vdash B$ type
3. [*ext*] $H \vdash C$ type
====
[*ext*] $H \vdash ((A \vee B) \to C) \to ((A \to C) \wedge (B \to C))$
BY autoT
1. [#]
....wf....
[*ext*] H; x: $(A \vee B) \to C \vdash A$ type
2. [#]
....main....
[*ext*] H; x: $(A \vee B) \to C$; x_1: $A \vdash C$
3. [#]
....wf....
[*ext*] H; x: $(A \vee B) \to C \vdash B$ type
4. [#]
....main....
[*ext*] H; x: $(A \vee B) \to C$; x_1: $B \vdash C$

Step 2

[*]
....main....
1. [*ext*] $H \vdash A$ type
2. [*ext*] $H \vdash B$ type
3. [*ext*] $H \vdash C$ type
====
[*ext*] H; x: $(A \vee B) \to C$; x_1: $A \vdash C$
BY dT 2 thenT autoT
1. [#]
....assertion....
[*ext*] H; x: $(A \vee B) \to C$; x_1: $A \vdash A \vee B$

Table 13.19: Proof steps taken by `autoT`

Address	Annotation	Goal
0	`[autoT]`	$H \vdash ((A \vee B) \Rightarrow C) \Rightarrow ((A \Rightarrow C) \wedge (B \Rightarrow C))$
0/0	`<compose>`	
0/0/0	`[dT 0]`	
0/0/0/0	`[Fol_implies.implies_intro [Hyp(1)] x]`	
0/0/1	`<compose>`	$H \vdash ((A \vee B) \Rightarrow C)$ *type*
0/0/1/0	`[dT 0]`	
0/0/1/0/0	`[Fol_implies.implies_type [Hyp(1)]]`	
0/0/1/1	`<compose>`	$H \vdash ((A \vee B)$ *type*
0/0/1/1/0	`[dT 0]`	
0/0/1/1/0/0	`[Fol_or.or_type [Hyp(1)]]`	
0/0/1/1/1	`<compose>`	$H \vdash A$ *type*
0/0/1/1/1/0	`[trivialT]`	
0/0/1/1/1/0/0	`[nthAssumT 0]`	
0/0/1/1/2	`<compose>`	$H \vdash B$ *type*
0/0/1/1/2/0	`[trivialT]`	
0/0/1/1/2/0/0	`[nthAssumT 1]`	
0/0/1/2	`<compose>`	$H \vdash C$ *type*
0/0/1/2/0	`[trivialT]`	
0/0/1/2/0/0	`[nthAssumT 2]`	
0/0/2	`<compose>`	$H,\ (A \vee B) \Rightarrow C \vdash (A \Rightarrow C) \wedge (B \Rightarrow C)$
0/0/2/0	`[dT 0]`	
0/0/2/0/0	`[Fol_and.and_intro [Hyp(2)]]`	
0/0/2/1	`<compose>`	$H,\ (A \vee B) \Rightarrow C \vdash A \Rightarrow C$
0/0/2/1/1	`<identity>`	$H,\ (A \vee B) \Rightarrow C \vdash A$ *type*
0/0/2/1/2	`<identity>`	$H,\ (A \vee B) \Rightarrow C,\ x_1{:}A \vdash C$
0/0/2/2	`<compose>`	$H,\ (A \vee B) \Rightarrow C \vdash B \Rightarrow C$
0/0/2/2/0	`[dT 0]`	
0/0/2/2/0/0	`[Fol_implies.implies_intro [Hyp(2)] x_1]`	
0/0/2/2/1	`<identity>`	$H,\ (A \vee B) \Rightarrow C \vdash B$ *type*
0/0/2/2/2	`<identity>`	$H,\ (A \vee B) \Rightarrow C,\ x_1{:}B \vdash C$

defined in the modules for each of the logical operators. The *resource annotations* on the rules provided the connection to the dT tactic. The trivialT tactic performed the "trivial" operations of proving the goal from an existing assumption.

Going back to the refinement performed by autoT, note that the first and third subgoals are instances of the assumptions, but they contain an extra hypothesis, which prevents the autoT from proving them automatically. We can augment the autoT tactic to recognize these cases (as it is done in the Nuprl type theory), but it will be simpler for this proof the do the task manually.

• *Prove the first and third subgoals with the tactic* `thinT 2 thenT trivialT`.

The remaining two goals are similar, we will prove subgoal two. In this case the autoT tactic was unable to make progress because the next likely step is an elimination step, and the default implementation of autoT in the module Base_auto_tactic does not perform elimination reasoning. Instead, we will perform the reasoning by hand.

• *Navigate to the second subgoal, and apply the* `dT 2 thenT autoT` *tactic (Table 13.18, Step 2).*

The dT 2 tactic performs elimination reasoning by producing two subgoals: one is the subgoal listed in the table, and the other is the subgoal $H,\ (A \vee B) \Rightarrow C,\ A,\ C \vdash C$, which is immediately provable by the autoT tactic.

The remaining subgoal is not proved by autoT because the application of dT 0 requires a *selection* argument to determine the branch of the disjunction. We can perform this step by hand.

• *Navigate to the subgoal, and apply the* `selT 1 (dT 0) thenT autoT` *tactic (Table 13.20, Step 3).*

This last step leaves only the trivial subgoal *B type*, which we can prove by thinning the irrelevant subgoals, and applying the trivialT tactic.

• *Prove the remaining subgoal using the* `thinT 3 thenT thinT 2 thenT trivialT` *tactic (Table 13.20, Step 4).*

This complete the proof of the distrib_or theorem.

13.1.9 Proof automation for constructive propositional logic

While the proof of the distrib_or theorem is not difficult, it still requires far too many steps of interaction. Many of these steps are obvious, and in fact we know that the truth of statements in the propositional logic are in fact decidable. For the final step in the propositional logic, we will develop a simple (although inefficient) proof procedure for propositional sentences, using Dyckhoff's algorithm [DP99].

Table 13.20: Prove the `distrib_or` theorem

Step 3
[*]
....assertion....
1. [*ext*] $H \vdash A$ type
2. [*ext*] $H \vdash B$ type
3. [*ext*] $H \vdash C$ type
====
[*ext*] H; x: $(A \vee B) \rightarrow C$; x_1: $A \vdash A \vee B$
BY selT 1 (dT 0) thenT autoT
1. [#]
....wf....
[*ext*] H; x: $(A \vee B) \rightarrow C$; x_1: $A \vdash B$ type

Step 4
* * * [*]
....wf....
1. [*ext*] $H \vdash A$ type
2. [*ext*] $H \vdash B$ type
3. [*ext*] $H \vdash C$ type
====
[*ext*] H; x: $(A \vee B) \rightarrow C$; x_1: $A \vdash B$ type
BY thinT 3 thenT thinT 2 thenT trivialT

Dyckhoff's algorithm is based on the use of three new rules for implication elimination, shown in Table 13.21. The algorithm is very simple: search all proof trees in some order, using the implication rules in Table 13.21 instead of `implies_elim` when possible. Dyckhoff's algorithm interprets negations as implications, and we will need to unfold all the definitions of negation in the goal sequent.

We will program this proof procedure as a tactic. For the first step, we need to add `open` statements for the modules that have functions we need to use.

• *Add the following* `open` *declarations to the* `fol_prop.ml` *file:*

```
open Refiner.Refiner.Term
open Refiner.Refiner.TermOp
open Refiner.Refiner.RefineError
open Tactic_type
open Tactic_type.Tacticals
open Tactic_type.Conversionals
open Base_dtactic
open Base_auto_tactic
open Fol_not
open Fol_struct
```

The `Refiner` modules define operations on MetaPRL *terms*; the `Tactic_type` modules define basic tacticals and conversionals; the `Base_dtactic` defines the `dT` tactic and the `Base_auto_tactic` defines the `trivialT` tactic; and we also need to `Fol_not` module to access the `unfold_not` rewrite, and the `Fol_struct` modules to access the `thinT` tactic.

For the next step, we will want to state and derive Dyckhoff's rules for implication elimination.

• *Add the rules and tactics in Table 13.21 to the* `fol_prop.ml` *file.*

The tactics define the standard wrappers for providing arguments to the rules. The proofs of the rules are straightforward and we omit the proofs here.

The next step is to define ML functions that test whether a term matches one of the special cases of implication. The `Refiner` module provides many operations for examining terms, including accessing the opname, the parameters, and the subterms of a term.

• *Add the term predicate functions in Table 13.23 to the* `fol_prop.ml` *file.*

In general, operations on terms require the *opname*, which can be computed with the `opname_of_term` function. An implication has opname `implies_opname`, and it has two subterms, each with no bindings

Table 13.21: Rules for implication elimination

```
(* Refinement of implication *)
interactive imp_and_rule 'H 'J 'u :
   [wf] sequent['ext]
      {'H; x: ('C & 'D) => 'B; 'J >- "type"{'C}} -->
   [wf] sequent['ext]
      {'H; x: ('C & 'D) => 'B; 'J >- "type"{'D}} -->
   [main] sequent['ext]
      {'H; 'J; u: 'C => 'D => 'B >- 'T} -->
   sequent['ext] {'H; x: ('C & 'D) => 'B; 'J >- 'T}

interactive imp_or_rule 'H 'J 'u 'v :
   [wf] sequent['ext]
      {'H; x: ('C or 'D) => 'B; 'J >- "type"{'C}} -->
   [wf] sequent['ext]
      {'H; x: ('C or 'D) => 'B; 'J >- "type"{'D}} -->
   [main] sequent['ext]
      {'H; 'J; u: 'C => 'B; v: 'D => 'B >- 'T} -->
   sequent['ext] {'H; x: ('C or 'D) => 'B; 'J >- 'T}

interactive imp_imp_rule 'H 'J 'u :
   [wf] sequent['ext]
      {'H; x: ('C => 'D) => 'B; 'J >- "type"{'C}} -->
   [wf] sequent['ext]
      {'H; x: ('C => 'D) => 'B; 'J >- "type"{'D}} -->
   [main] sequent['ext]
      {'H; 'J; u: 'D => 'B >- 'T} -->
   sequent['ext] {'H; x: ('C => 'D) => 'B; 'J >- 'T}
```

Table 13.22: Tactics for implication elimination

```
let d_and_impT i p =
   let u = Var.maybe_new_vars1 p "u" in
   let j, k = Sequent.hyp_indices p i in
      imp_and_rule j k u p

let d_or_impT i p =
   let u, v = Var.maybe_new_vars2 p "u" "v" in
   let j, k = Sequent.hyp_indices p i in
      imp_or_rule j k u v p

let d_imp_impT i p =
   let u = Var.maybe_new_vars1 p "u" in
   let j, k = Sequent.hyp_indices p i in
      imp_and_rule j k u p
```

(by naming traditions, it is a dep0_dep0 term because it has two subterms, each with 0 binding variables). The function is_imp_and_term $(A \wedge B) \Rightarrow C$ returns *true* because the outermost operator is an implication, and the first subterm $A \wedge B$ is a conjunction.

For the final step we write the code that performs the proof search.

• *Provide a proof search algorithm by adding the code in Table 13.24 to the* fol_prop.ml *module.*

The proof search algorithm has two parts: the decompPropDecideHypT applies an elimination rule, and the decompPropDecideConclT applies an introduction rule. Both functions perform a case analysis first to select the rule to apply. The ifNotWT function is a tactical that only applies its arguments to goals that are *not* marked [wf]. The product of this search code is the propDecideT function, which first unfolds all occurrences of negations in all clauses of the sequent, and then applies the internal version of the search procedure.

Let's see how this works.

• *Make a copy of the* distrib_or *rule statement as a new rule with name* distrib_or2*. Edit it, and apply the* propDecideT *tactic (Table 13.25).*

The statement is essentially proved—all that remains are well-formedness subgoals. To take care of these, we can provide a quick "hack" to prove subgoals of this form. (the Itt_logic module defines a more general procedure).

• *Add a tactic to prove well-formedness subgoals (Table 13.26).*

Table 13.23: Term predicate functions for testing implication

```
let false_opname = opname_of_term << "false" >>
let is_false_term = is_no_subterms_term false_opname

let and_opname = opname_of_term << 'A & 'B >>
let is_and_term = is_dep0_dep0_term and_opname

let or_opname = opname_of_term << 'A or 'B >>
let is_or_term = is_dep0_dep0_term or_opname

let implies_opname = opname_of_term << 'A => 'B >>
let is_implies_term =
   is_dep0_dep0_term implies_opname

let is_imp_and_term term =
   is_implies_term term
   & is_and_term (fst (two_subterms term))

let is_imp_or_term term =
   is_implies_term term
   & is_or_term (fst (two_subterms term))

let is_imp_imp_term term =
   is_implies_term term
   & is_implies_term (fst (two_subterms term))
```

Table 13.24: Proof search algorithm

```
(* Try to decompose a hypothesis *)
let rec decompPropDecideHypT i p =
  (let _, term = Sequent.nth_hyp p i in
     if is_false_term term then
       dT i
     else if is_and_term term or is_or_term term then
       dT i thenT ifNotWT internalPropDecideT
     else if is_imp_and_term term then
       (* {C & D => B} => {C => D => B} *)
       d_and_impT i thenT ifNotWT internalPropDecideT
     else if is_imp_or_term term then
       (* {C or D => B} => {(C => B) & (D => B)} *)
       d_or_impT i thenT ifNotWT internalPropDecideT
     else if is_imp_imp_term term then
       (* {(C => D) => B} => {D => B} *)
       d_imp_impT i thenT ifNotWT internalPropDecideT
     else if is_implies_term term then
       dT i thenT thinT i thenT ifNotWT internalPropDecideT
     else
       (* Nothing recognized, try to see if we're done. *)
       nthHypT i) p

(* Decompose the goal *)
and decompPropDecideConclT p =
  (let goal = Sequent.concl p in
     if is_or_term goal then
       (selT 1 (dT 0) thenT ifNotWT internalPropDecideT)
       orelseT (selT 2 (dT 0)
                thenT ifNotWT internalPropDecideT)
     else if is_and_term goal or is_implies_term goal then
       dT 0 thenT ifNotWT internalPropDecideT
     else
       trivialT) p

and internalPropDecideT p =
  (onSomeHypT decompPropDecideHypT
     orelseT decompPropDecideConclT) p

let propDecideT =
  onAllClausesT (fun i -> tryT (rw (sweepUpC unfold_not) i))
  thenT internalPropDecideT
```

Table 13.25: Applying the `propDecideT` tactic to `distrib_or`

[*]
....main....
1. [*ext*] $H \vdash A$ type
2. [*ext*] $H \vdash B$ type
3. [*ext*] $H \vdash C$ type
====
[*ext*] $H \vdash ((A \vee B) \rightarrow C) \rightarrow ((A \rightarrow C) \wedge (B \rightarrow C))$
BY propDecideT
1. [#]
....wf....
[*ext*] $H \vdash ((A \vee B) \rightarrow C)$ type
2. [#]
....wf....
[*ext*] H; x: $(A \vee B) \rightarrow C \vdash A$ type
3. [#]
....wf....
[*ext*] H; x: $(A \vee B) \rightarrow C \vdash B$ type
4. [#]
....wf....
[*ext*] H; u: $A \rightarrow C$; v: $B \rightarrow C \vdash A$ type
5. [#]
....wf....
[*ext*] H; u: $A \rightarrow C$; v: $B \rightarrow C \vdash B$ type

Table 13.26: A simple tactic for proving well-formedness subgoals

```
let thinAllT i j p =
   let rec tac j =
      if j < i then
         idT
      else
         thinT j thenT tac (pred j)
   in
      tac j p

let proveTypeT p =
   (thinAllT 2 (Sequent.hyp_count p) thenT trivialT) p
```

Table 13.27: The Fol_pred module

```
include Fol_type
open Tactic_type
open Tactic_type.Tacticals
open Base_auto_tactic

(* Syntax and display *)
declare pred
dform pred_df : pred = ‘"Pred"

(* Type judgment *)
prim pred_type 'H 'J :
   sequent ['ext] { 'H; x: pred; 'J['x] >- "type"{'x} }
   = trivial

let d_pred_type i p =
   let j, k = Sequent.hyp_indices p i in
      pred_type j k p

let trivial_resource =
   Mp_resource.improve trivial_resource (**)
      { auto_name = "d_pred_type";
        auto_prec = trivial_prec;
        auto_tac = onSomeHypT d_pred_type
      }
```

With this tactic, the distrib_or2 theorem is provable in one step with the tactic propDecideT thenT autoT thenT proveTypeT.

13.1.10 The quantifiers

To complete the definition of the first-order logic, we need to add the quantifiers $\forall$ and $\exists$. Their definitions are much like the propositional connectives with one exception: they require a binding mechanism. The informal meaning of a formula $\forall x.B[x]$ is that for any *predicate* x, the predicate $B[x]$ is true. To implement the quantifiers, we first need to provide a module Fol_pred that defines a term standing for all predicates.

• *Define the term for predicates with the code in Table 13.27.*

Predicates are, by definition, types. We also add this inference to

the `trivialT` tactic.

Next, we will define a module for the universal quantifier.

• *Define the module for* `Fol_all` *with the code in Table 13.28.*

The `all_type` and `all_intro` rules use quantifications over predicates. The `all_elim` rule uses a new option for the `elim_resource` that specifies that the hypothesis $\forall y.B[y]$ should be thinned by default using the `thinT` tactic. Otherwise, there is little that is new in these definitions. The `Fol_exists` module is similar, and we omit it definition (a version can be found in the `~/meta-prl/fol/fol_exists.ml` file.

As the final step in the defining the (constructive) first-order logic, we collect all the modules into a single module `Fol_theory`.

• *Define the constructive first-order logic with the code in Table 13.29.*

The `Fol_theory` module includes all of the modules in the first-oder logic, and it also includes *relations* between the display precedences of the display forms for each of the operators.

13.2 Relating the propositional logic to type theory

So far, we have developed the first-order logic as an isolated theory without any relations to any of the other logics in MetaPRL. In this section, we'll see how to relate the propositional fragment of the logic to the Nuprl type theory. The correspondence is guided by the Curry-Howard isomorphism, which gives a intepretation of each logical operator as a type constructor.

The *mechanism* for providing a derivation uses both the `include` and `derive` statements in the MetaPRL module system. There are three basic steps in a derivation of a logical operator:

1. `include` the Nuprl type theory,
2. state the *interpretations* of the logical operators using the `rewrite` mechanism,
3. `derive` the rules for the logical operator.

We'll start by deriving the `Fol_type` module with the following code. The first step includes the `Itt_theory` module, which defines the Nuprl type theory. The next step, `derive Fol_type` states that the `Fol_type` module is to be *derived*, and that all its rules become proof obligations. The `Fol_type` module has no rules, so in this case there are no extra obligations. The final statement, a `prim_rw` definition, provides the definition of the type judgment in the `Fol_type` module directly in terms of the type judgment in the `Itt_equal` module in the Nuprl type theory.

Table 13.28: Definition of the universal quantifier

```
include Fol_implies
include Fol_struct
include Fol_pred

open Base_dtactic
open Fol_struct

declare "all"{x. 'B['x]}

prec prec_all

dform all_df : parens :: "prec"["prec_all"] ::
   "all"{x. 'B} =
   szone pushm[3] forall slot{'x} `"."
   hspace slot{'B} popm ezone

prim all_type {| intro_resource [] |} 'H 'x :
   [wf] sequent ['ext]
      { 'H; x: pred >- "type"{'B['x]} } -->
   sequent ['ext]
       { 'H >- "type"{."all"{y. 'B['y]}} }
       = trivial

prim all_intro {| intro_resource [] |} 'H 'x :
   [main] ('b['x] : sequent ['ext]
      { 'H; x: pred >- 'B['x] }) -->
   [wf] sequent ['ext]
      { 'H; x: pred >- "type"{'B['x]} } -->
   sequent ['ext] { 'H >- "all"{y. 'B['y]} } =
   lambda{y. 'b['y]}

prim all_elim
 {| elim_resource [ThinOption thinT] |} 'H 'J 'x 'z 'a :
 [wf] sequent ['ext]
  { 'H; x: "all"{y. 'B['y]}; 'J['x] >- "type"{'a} } -->
 [main] ('b['x; 'z] : sequent ['ext]
  { 'H; x: "all"{y. 'B['y]};
    'J['x]; z: 'B['a] >- 'C['x] }) -->
 sequent ['ext] { 'H; x: "all"{y. 'B['y]};
    'J['x] >- 'C['x] } =
 'b['x; 'x 'a]
```

Table 13.29: The first-order logic Fol_theory

```
include Fol_type
include Fol_false
include Fol_true
include Fol_and
include Fol_or
include Fol_implies
include Fol_not
include Fol_struct
include Fol_pred
include Fol_all
include Fol_exists

open Fol_implies
open Fol_and
open Fol_or
open Fol_not
open Fol_all
open Fol_exists

prec prec_implies < prec_and
prec prec_implies < prec_or
prec prec_or < prec_and
prec prec_and < prec_not
prec prec_all < prec_implies
prec prec_exists < prec_implies
```

Table 13.30: The interpretation of the `false` constant

```
include Itt_theory
include Fol_itt_type

derive Fol_false

prim_rw unfold_false : "false" <--> void

derive false_type 'H :
   sequent ['ext] { 'H >- "type"{."false"} }

derive false_elim 'H 'J :
   sequent ['ext] { 'H; x: "false"; 'J['x] >- 'C['x] }
```

```
include Itt_theory
derive Fol_type
prim_rw unfold_type : Fol_type!"type"{'t} <-->
    Itt_equal!"type"{'t}
```

The logical operators are a little more interesting. To derive the `Fol_false` module, we need to give an interpretation of the logical constant `false`. In keeping with the Curry-Howard isomorphism, the constant `false` should be represented by an *empty* type (there are no proofs of `false`). The code for the `Fol_itt_false` module is shown in Table 13.30.

The `Fol_itt_type` module is included for the definition of the `type` judgment, and our goal is to derive the module `Fol_false`. The rewrite gives the definition of `false` in terms of the `void` type from the module `Itt_void` (in this case the unqualified names are unambiguous). The next step is to derive the rules from the `Fol_false` module.[1] To the editor, the `derive` statement for a rule is treated the same as if the rule were declared as `interactive`: it determines a proof obligation that must be satisfied with the editor. So the next question is: how do we prove the `false_type` and `false_elim` rules?

In general, the first step in a derivation is to unfold the definitions of the defined terms—in this case `Fol_type!type` and `Fol_false!false`. The proof is shown in Table 13.31.

[1] At present, the derivations must be stated explicitly. This may go away in future version of the system, since the derivations may be inferred from the `derive Fol_false` statement.

Table 13.31: Proving the `false_type` rule

Step 1
| # [*]
| ....main....
| [*ext*] $H\vdash$ False type
| **BY** rwh unfold_false 0 thenT rwh unfold_type 0
| **1. [#]**
| ....main....
| [*ext*] $H\vdash$ Void Type

Step 2
| * * [*]
| ....main....
| [*ext*] $H\vdash$ Void Type
| **BY** dT 0

Table 13.32: Constructive interpretation of the logical operators

Operator	Interpretation
`type{'t}`	`Itt_equal!type{'t}`
`false`	`Itt_void!void`
`true`	`Itt_unit!unit`
`and{'A; 'B}`	`Itt_prod!prod{'A; 'B}`
`or{'A; 'B}`	`Itt_union!union{'A; 'B}`
`implies{'A; 'B}`	`Itt_fun!fun{'A; 'B}`

Step 1 replaces the definition of `false` with `void` and `Fol_type!type` (which displays the term `type`$\{t\}$ as t `type`) with `Itt_equal!type` (which displays the term `type`$\{t\}$ as t `Type`). The Nuprl type theory is automated in much the same way as we provided the automation for the first-order logic, and we can use the `dT 0` tactic to knock off the subgoal, as shown in Step 2.

The interpretation for all the logical operators is shown in Table 13.32. The constant `true` is interpreted as the type `unit`, which contains exactly one proof $\cdot$. For the propositional operators, `and` is interpreted as a Cartesian product, `or` as a disjoint union, and `implies` as a function space.

The modules for the propositional operators are all similar. The `Fol_and` module, listed in Table 13.33, shows the common outline. The

first part establishes that this is a derivation from `Itt_theory` and `Fol_itt_type` modules to the module `Fol_and`.

The rewrites provide the definitions of the operators: the conjuction is the Cartesian product, and the proof terms have the corresponding representations. Rewrite definitions define conversions that operate from left-to-right: the `unfold_and` conversion rewrites a conjunction to a product space. It is also useful to have the right-to-left version, called `fol_and`, that rewrites the product space to the conjunction. The `makeFoldC` conversional (defined in `Tactic_type.Conversionals` build a conversion for right-to-left rewriting given the redex term and the rewrite definition.

The last part of the `Fol_itt_and` module states the rules that are to be derived. To make an arbitrary choice, let's look at the proof of the theorem `and_type`. The initial step is to unfold the definitions for the conjuction and type judgment, shown in Table 13.34, Step 1. This refinement produces two similar subgoals, both of which are similar (but not identical to) the assumptions.

The proof of one of these subgoals requires *folding* the definition of the `type` judgment term. It also requires an `unsquashT` operation: the optional argument to the sequent in the subgoal is the term `squash` $(\cdot)$, rather than the conventional `'ext` variable, and the `unsquashT` tactic will perform the replacement. The proof for the first subgoal is shown in Step 2.

Unfortunately, this interpretation breaks down for the quantifiers because the first-order logic is *impredicative* and the type theory is *predicative*. Briefly, this means that quantification in the type theory is strictly inductively defined, and there is no non-trivial type that includes itself as a member. In contrast, in the first-order logic, a predicate like $\forall P.P \Rightarrow P$ is quantified over *all* predicates including the predicate itself.

Instead, we will leave the derivation as it stands (for the propositional logic only), and we will skip to developing a *classical* first-order logic.

13.3 Defining classical first-order logic

Classical logic is a conservative extension to the constructive logic we have just defined (all the existing rules are still valid). In a classical setting, every predicate is either *true* or *false*. We can enable classical reasoning in this single-conclusion logic with a single rule, based on double-negation elimination $\neg\neg P \Rightarrow P$. The proof term for this proposition has no computational meaning, so we provide a new term `magic` to represent proofs by excluded middle.

Table 13.33: Listing of the Fol_itt_and module

```
include Itt_theory
include Fol_itt_type

derive Fol_and

prim_rw unfold_and : "and"{'A; 'B} <--> prod{'A; 'B}
prim_rw unfold_pair : Fol_and!pair{'a; 'b} <-->
    Itt_dprod!pair{'a; 'b}
prim_rw unfold_spread :
    Fol_and!spread{'a; x, y. 'b['x; 'y]} <-->
    Itt_dprod!spread{'a; x, y. 'b['x; 'y]}

let fold_and =
   makeFoldC << Fol_and!"and"{'A; 'B} >> unfold_and
let fold_pair =
   makeFoldC << Fol_and!"pair"{'a; 'b} >> unfold_pair
let fold_spread =
   makeFoldC << Fol_and!spread{'a; x, y. 'b['x; 'y]} >>
   unfold_spread

derived_rw reduce_spread :
   spread{pair{'x; 'y}; a, b. 'body['a; 'b]} <-->
   'body['x; 'y]

derived and_type 'H :
   [wf] sequent ['ext] { 'H >- "type"{'A} } -->
   [wf] sequent ['ext] { 'H >- "type"{'B} } -->
   sequent ['ext] { 'H >- "type"{.Fol_and!"and"{'A; 'B}} }

derived and_intro 'H :
   [main] ('a : sequent ['ext] { 'H >- 'A }) -->
   [main] ('b : sequent ['ext] { 'H >- 'B }) -->
   sequent ['ext] { 'H >- Fol_and!"and"{'A; 'B} }

derived and_elim 'H 'J 'x 'y 'z :
   [main] ('body['y; 'z] : sequent ['ext]
      { 'H; y: 'A; z: 'B; 'J[Fol_and!pair{'y; 'z}] >-
        'C[Fol_and!pair{'y; 'z}] }) -->
   sequent ['ext]
      { 'H; x: Fol_and!"and"{'A; 'B}; 'J['x] >- 'C['x] }
```

Table 13.34: Prove the `and_type` theorem

Step 1
* [*]
....main....
1. [*ext*] $H \vdash A$ type
2. [*ext*] $H \vdash B$ type
====
[*ext*] $H \vdash (A \wedge B)$ type
BY rwh unfold_and 0 thenT rwh unfold_type 0 thenT dT 0
1. [*]
....wf....
[·] $H \vdash A$ Type
2. [*]
....wf....
[·] $H \vdash B$ Type

Step 2
* * [*]
....wf....
1. [*ext*] $H \vdash A$ type
2. [*ext*] $H \vdash B$ type
====
[·] $H \vdash A$ Type
BY rwh fold_type 0 thenT unsquashT « 'ext » thenT trivialT

Table 13.35: Providing classical reasoning

```
include Fol_not

open Tactic_type

declare magic{x. 't['x]}

dform magic_df : magic = `"magic"

prim magic 'H 'x :
   ('t['x] : sequent ['ext]
       { 'H; x: "not"{'T} >- "false" }) -->
   sequent ['ext] { 'H >- 'T } =
   magic{x. 't['x]}

let magicT p =
   let v = Var.maybe_new_vars1 p "v" in
      magic (Sequent.hyp_count_addr p) v p
```

The implementation `fol_magic.ml` is shown in Table 13.35. The module `Fol_not` is required for the definition of negation. The `magic` rule essentially states the following: to prove a predicate T, assume that T is false and prove a contradiction. The `magicT` tactic provides the standard tactic wrapping functionality.

The classical theory can be pulled together as a module with just two lines that include the constructive version of the logic tegether with the `magic` module.

```
include Fol_theory
include Cfol_magic
```

13.4 Deriving classical first-order logic from the Nuprl type theory

Deriving the classical logic from type theory is significantly different from deriving the constructive logic. The main obstacle is that the Nuprl type theory is constructive, and there is no way to justify the `magic` rule directly. Instead, we will build a *model* of classical logic in the type theory, rather than coding it directly using the most obvious type constructors.

Another obstacle is that in the classical system, every predicate is either *true* or it is *false*, and it is natural to consider two predicates to be equal when they are either both true or both false.

Initially, we can start along the same path as the interpretation for the constructive logic. First, we intend to use the types `void` and `unit` as the canonical representatives for `false` and `true`. However, if we make these choices, we will have problems interpreting the propositional connectives. For example, the obvious interpretation of $A \wedge B$ is the product space $A \times B$. But this breaks equality reasoning: $\texttt{true} \wedge \texttt{true} = \texttt{unit} \times \texttt{unit} \neq \texttt{true}$. In some sense, the problems arises because of the computational content of the types: a product space contains *pairs*, while the `unit` type contains only the canonical term $\cdot$.

To solve this problem, we use a standard Nuprl technique for *computational hiding*. The $\texttt{equash}\{T\}$ term is a type for any type T, and it contains the unique canonical element $\cdot$ if T is inhabited, and it is empty otherwise. Furthermore, $\texttt{esquash}\{T_1\} = \texttt{esquash}\{T_2\}$ whenever $T_1 \Leftrightarrow T_2$. The usual display notation for $\texttt{equash}\{T\}$ is $\downarrow T$.

The initial starting point for the interpretation is then as follows. The interpretation of `false` is the type $\downarrow \texttt{void}$ and the interpretation of `true` is $\downarrow \texttt{unit}$. The $\texttt{type}\{P\}$ judgment enforces the condition that the predicate P be either true or false with the interpretation $\downarrow (P = \mathit{true} \in \mathbb{U}_1 \vee P = \mathit{false} \in \mathbb{U}_1)$.

We will define this initial part of the interpretation in the module `Cfol_itt_base`. The module definition is shown in two parts, in tables 13.36 and 13.37. The derivation is for the four modules `Fol_type`, `Fol_false`, `Fol_true`, and `Fol_pred` all at once. The definitions for false and true are trhe squashed versions of void and unit, and the type judgment is the squashed disjunction stating that the argument predicate is either false or true. The `pred` type is the set of types that are predicates. The next few theorems are conventient lemmas stating some relationships between the predicates in the types in `univ[1:l]`.

The next step is to derives the rules for `false`, `true`, and `pred`. These proofs are fairly strightforward. Note that in this interpretation we get the additional facts that `true` and `false` are members of the `pred` type.

The interpretation of the logical operators is shown in Table 13.39. The proofs are fairly long. The `all_elim` proof illustrates the kinds of reasoning involved; Table 13.40 shows the initial goal in the proof. The basic strategy is as follows: first, $\forall v.B[v]$ is equivalent to $B[\mathit{false}] \wedge B[\mathit{true}]$. Since a is either true or false, then $B[a]$ is equivalent to either $B[\mathit{false}]$ or $B[\mathit{true}]$, so $B[a]$ follows by assumption, and we can add it to the sequent using the `cut` rule.

Table 13.36: Base interpretation Cfol_itt_base (Part 1)

```
include Itt_theory

derive Fol_type
derive Fol_false
derive Fol_true
derive Fol_pred

open Refiner.Refiner.TermSubst
open Tactic_type
open Tactic_type.Conversionals
open Itt_struct
open Itt_equal

(* Interpretation *)
prim_rw unfold_false : Fol_false!"false" <-->
    esquash{void}
prim_rw unfold_true : Fol_true!"true" <-->
    esquash{unit}
prim_rw unfold_type : Fol_type!"type"{'t} <-->
   esquash{(('t = "false" in univ[1:l])
            or ('t = "true" in univ[1:l]))}
prim_rw unfold_pred : Fol_pred!"pred" <-->
   { T: univ[1:l] | "type"{'T} }

(* Lemmas *)
interactive false_univ {| intro_resource [] |} 'H :
   sequent ['ext]
      { 'H >- member{univ[1:l]; ."false"} }

interactive true_univ {| intro_resource [] |} 'H :
   sequent ['ext]
      { 'H >- member{univ[1:l]; ."true"} }

interactive type_univ 'H :
   [wf] sequent ['ext] { 'H >- "type"{'A} } -->
   sequent ['ext]
      { 'H >- member{univ[1:l]; 'A} }

let typeT p =
   type_univ (Sequent.hyp_count_addr p) p
```

Table 13.37: Base interpretation Cfol_itt_base (Part 2)

```
(* Rules for false *)
derived false_type 'H :
   sequent ['ext] { 'H >- "type"{."false"} }

derived false_elim 'H 'J :
   sequent ['ext] { 'H; x: "false"; 'J['x] >- 'C['x] }

(* Rules for true *)
derived true_type 'H :
   sequent ['ext] { 'H >- "type"{."true"} }

derived true_intro 'H :
   sequent ['ext] { 'H >- "true" }

interactive true_concl_elim 'H :
   [main] sequent ['ext]
      { 'H >- 'P = "true" in univ[1:l] } -->
   sequent ['ext] { 'H >- 'P }

interactive true_concl_intro 'H :
   [wf] sequent ['ext] { 'H >- "type"{'P} } -->
   [main] sequent ['ext] { 'H >- 'P } -->
   sequent ['ext] { 'H >- 'P = "true" in univ[1:l] }

let trueT p =
 let goal = Sequent.concl p in
   if is_equal_term goal then
     true_concl_intro (Sequent.hyp_count_addr p) p
   else
     true_concl_elim (Sequent.hyp_count_addr p) p
```

Table 13.38: Base interpretation Cfol_itt_base (Part 3)

```
(* Rules for pred *)
interactive pred_elim {| elim_resource [] |} 'H 'J 'y :
   sequent ['ext]
      { 'H; x: univ[1:l]; y: "type"{'x}; 'J['x]
      >- 'C['x] } -->
   sequent ['ext] { 'H; x: pred; 'J['x] >- 'C['x] }

interactive true_pred {| intro_resource [] |} 'H :
   sequent ['ext] { 'H >- member{pred; ."true"} }

interactive false_pred {| intro_resource [] |} 'H :
   sequent ['ext] { 'H >- member{pred; ."false"} }

interactive pred_type1 {| intro_resource [] |} 'H :
   sequent ['ext] { 'H >- Itt_equal!"type"{pred} }

derived pred_type 'H 'J :
   sequent ['ext] { 'H; x: pred; 'J['x] >- "type"{'x} }
```

Table 13.39: Interpretation of the classical operators

false	esquash{void}
true	esquash{unit}

type{'t}	esquash{
	union{'t = false in univ[1:l];
	't = true in univ[1:l]}}

pred	{ T: univ[1:l] | type{'T} }
and{'A; 'B}	esquash{prod{'A; 'B}}
or{'A; 'B}	esquash{union{'A; 'B}}
implies{'A; 'B}	esquash{fun{'A; 'B}}
all{v. 'B['v]}	and{'B[false]; 'B[true]}
exists{v. 'B['v]}	or{'B[false]; 'B[true]}

Table 13.40: Prove the `all_elim` theorem

Step 1
! [**!**]
....main....
1. [*ext*] H; x: $\forall y.\ B[y]$; $J[x]\vdash\ a$ type
2. [*ext*] H; x: $\forall y.\ B[y]$; $J[x]$; z: Pred$\vdash\ B[z]$ type
3. [*ext*] H; x: $\forall y.\ B[y]$; $J[x]$; z: $B[a]\vdash\ C[x]$
====
[*ext*] H; x: $\forall y.\ B[y]$; $J[x]\vdash\ C[x]$
BY assertT « 'B['a] » thenMT autoT
1. [!]
....main....
[*ext*] H; x: $\forall y.\ B[y]$; $J[x]\vdash\ B[a]$

13.5 Conclusion

This concludes definition of the first-order logic. We have seen:

- how to define a primitive logic, include constructive and classical versions of first-order logic,
- how to add automation and decisions procedures to a logic,
- how to use resources to simplify the task of adding proof automation,
- how to define a theory *relation*—in particular how to derive the first-order logics from the Nuprl type theory.

First-order logic is a fairly simple logic; the definition of the logic and its automation is fairly straightforward. The next Chapter describes the Nuprl type theory, which is a more sophisticated logic with much more proof automation. The CZF logic (Chapter 15) also serves as an example of defining a large theory and providing a theory relation.

Chapter 14

The Nuprl type theory

The Nuprl type theory is as Martin–Löf-style type theory [ML84a, ML82, ML75]. The Nuprl type theory significantly extends Martin–Löf's work, including:

- propositions-as-types (the Curry-Howard isomorphism),
- a theory of canonical forms, and an evaluation procedure that produces at most one canonical form,
- a typed equality-type $t_1 = t_2 \in T$ that respects evaluation in all three parts, and requires that T is a type, and that t_1 and t_2 are equal programs having type T,
- a semantics of sequents based on pointwise-functionality, and more.

We discuss the basic attributes of the type theory here. A description of the type theory of Nuprl1 can be found in the Nuprl book [CAB+86]; Constable describes more recent versions in several articles [Con92, Con98, Con00]. Allen [All86] gives another account of the type theory and its semantics.

The MetaPRL type theory differs from the Nuprl type theory in several significant ways.

- There is an additional judgment T *type* that states that T is a type.
- The treatment of omitted computational content (the "squashed" propositions defined in Section 14.2) is different. MetaPRL sequents include a *tag* that specifies whether the computational content of the proof is to be used.
- In MetaPRL, level expressions i for the type universes $\mathbb{U}_i$ (Section 14.3) are universally quantified for each inference rule. In

Nuprl they are universally quantified at the level of the entire type theory. In other words, two theorems in MetaPRL that use the same level expression i refer to *different* levels, while in Nuprl they are the same.

The type theory is concerned with two basic concepts: *programs* and *types*. The types classify the programs, and programs can compute types as well as other values. The operational semantics of programs is described using computional equivalences (or *rewrites*), and type judgments are described using a sequent calculus.

14.1 Overview of the Nuprl type theory

The types in the type theory are defined inductively. There are several *base* types, including

- *Void*: the empty type,
- *Unit*: the type containing exactly one element $\cdot$,
- *Atom*: the type of finite strings of characters,
- $\mathbb{Z}$: the integers,
- an order $i < j$ which is a type for any two integers i and j,
- a ternary equality, of the form $x = y \in T$ where T is a type, and x and y are objects of type T,
- the type "universes" $\mathbb{U}_i$, where a type in $\mathbb{U}_i$ also belongs to $\mathbb{U}_{i+1}$.

If an equality type is inhabited, it is solely inhabited by the term "$\cdot$" (the same term in the *Unit* type). The $\mathbb{Z}$ type is defined as a primitive base type, although as we will see, the integers can be defined as a recursive data type. Given the integers, the *Unit* type can be defined as $0 = 0 \in \mathbb{Z}$ and *Void* as $0 = 1 \in \mathbb{Z}$, although it is more convenient to define them as primitive (so that the integers can be derived later). Using the propositions-as-types correspondence, *True* can be defined as *Unit*, and *False* as *Void*. Similarly, if the order relation is inhabited, it contains the single object $\cdot$.

The types $\mathbb{U}_i$ are indexed over the natural numbers, and form a type hierarchy of *universes* that are monotonically increasing in type inclusion. The Nuprl type theory is *predicative*, as is the standard Martin–Löf type theory.

There are several type constructors that are used to construct composite types. Given two types T_1 and T_2, the *disjoint union* type can be constructed as $T_u \equiv T_1 + T_2$. Inhabitants of the disjoint union are tagged according to which branch of the union they belong to. If $x \in T_1$, then $inl(x) \in T_1 + T_2$, and if $y \in T_2$ then $inr(x) \in T_1 + T_2$. This corresponds to the constructive disjunction $T_1 \vee T_2$.

The dependent product provides a pairing construction similar to a Cartesian product. The term $x{:}T_1 \times T_2$ is a type if T_1 is a type, and T_2 is a type for any inhabitant x of T_1. The inhabitants of the product are pairs $\langle x, y \rangle$ where $x \in T_1$, and $y \in T_2(x)$. The dependency serves two purposes. First of all, it is possible to give more precise specifications of some constructive objects. For instance, we may represent a date by its year, its month, and the day of the month, so that the date would inhabit the type $\mathbb{N} \times \{1 \ldots 11\} \times \{1 \ldots 31\}$. However, obviously illegal dates would inhabit this type (February 31, for instance). We can give a more precise specification using the dependent type $\mathbb{N} \times month{:}\{1 \ldots 11\} \times \{1 \ldots days_per_month(month)\}$, where *days_per_month* is a function with the obvious definition.

Another use of the dependent type is for existential quantification. The type $x{:}T_1 \times T_2$ corresponds to the quantification $\exists x{:}T_1.T_2$. If T_2 does not depend on x, we get the conjunction $T_1 \wedge T_2$.

A variant of the product type is the *set* type, which is denoted by $\{x{:}T_1 \mid T_2\}$. This corresponds to the type $x{:}T_1 \times T_2$, except that the inhabitants of the set type are elements of T_1; the inhabitant of T_2 is discarded. However, $T_2(x)$ must have *some* inhabitant for any element $x \in \{x{:}T_1 \mid T_2\}$. That is, the type $\{x{:}T_1 \mid T_2\}$ is the subtype of T_1 where T_2 holds on every element. In certain parts of a proof it may be possible to use the fact that T_2 holds without needing the inhabitant (for instance, when an equality is being proved). In other cases it is possible to derive the inhabitant, for instance for an equality or order relation (the inhabitant is always $\cdot$). One of the immediate uses of the set to is to express subranges of the integers:

$$\begin{aligned} \{i \ldots j\} &\equiv \{k{:}\mathbb{Z} \mid i \le k \wedge k \le j\} \\ \{i \ldots\} &\equiv \{j{:}\mathbb{Z} \mid i \le j\} \\ \{\ldots i\} &\equiv \{j{:}\mathbb{Z} \mid j \le i\} \end{aligned}$$

Another important type constructor is the dependent function type. If T_1 is a type, and T_2 is a type given $x \in T_1$, then $x{:}T_1 \to T_2$ is a type that corresponds roughly to the function space with domain T_1 and range $T_2(x)$ for argument $x \in T_1$. Again the dependencies allow functions to be more precisely specified than the undependent type. For instance, all functions that return a day-of-month given a month as a

argument belong to $month\colon\{1\ldots11\}\rightarrow\{1\ldots days_per_month(month)\}$. According to propositions-as-types, the function type corresponds to universal quantification. The function type above corresponds to the quantification $\forall x\colon T_1.T_2$. If T_2 does not depend on x, then this becomes the normal implication $T_1\Rightarrow T_2$.

The final type constructor we will discuss is the recursive type definition. For a complete exposition of the constructor, see Paul Mendler's thesis [Men87]. If $T(X)$ is a type for all types X and if T is *monotonic* in X, then $\mu\,(X.T)$ is a type, and it has a *fixed point* $\mu\,(X.T)=T(\mu\,(X.T))$. The parameterized version is $\mu\,(y,X.T;z)$, which is a type if there is some type Z where z inhabits Z, and for any inhabitant y of Z, and type function $X\colon Z\rightarrow\mathbb{U}_i$, T is a type that is monotonic in X.

The inhabitants of the recursive types are those elements that inhabit the finite unrollings of the type. For instance, a list of ascending integers has the type $L_>(0)$ where

$$L_>(k)\equiv\mu\,(i,X.Unit+j\colon\{i\ldots\}\times X(j+1);k)$$

This type is inhabited by $inl\,\cdot$, since $inl\,\cdot\in Unit+j\colon\{i\ldots\}\times L_>(j+1)$ Similarly, we can show $inr\,\langle 5,inl\,\cdot\rangle$ and $inr\,\langle 4,inr\,\langle 5,inl\,\cdot\rangle\rangle$ inhabit the type, but not $inr\,\langle 5,inr\,\langle 4,inl\,\cdot\rangle\rangle$, nor $inr\,\cdot$.

Since the recursive definition has a fixed-point semantics, it is necessary to require that the term defining the type (T in $\mu\,(y,X.T;z)$) be monotone over the type argument (X). It is also necessary that the definition argument be a well-defined type for any value of the type argument. That is, the type argument does not range over unrolling of the type definition, it ranges over all types, and so it is not possible to impose any structure on inhabitants of the type denoted by the type argument. This makes it impossible to define a type that depends on the inhabitant of the type argument in a non-trivial way.

14.1.1 Nuprl proof system

When we discuss terms in a type theory like Nuprl it is generally necessary to distinguish between *terms*, which are just notation, and meaningful mathematical objects. For instance, $1+\mathbb{Z}$ is a term, but is not a meaningful mathematical object, and it does not belong to any type. In Nuprl the discourse about mathematical objects is limited to those objects that can be given a type. The semantics of the type theory is such that any term that can be given a type has a meaning and corresponds to some mathematical object. On the other hand, since the type theory is necessarily incomplete, if a term does not have a type it may or may not correspond to a meaningful mathematical object. In a Martin–Löf-

style type theory, it is generally undecidable which terms can be given a type, and what the type is. In Nuprl the decision was made to integrate type-checking with theorem proving. That is, when we prove a theorem, we must also prove that the theorem statement is a type. This is an incremental process, and a large part of the proof rules in Nuprl are concerned with checking the *well-formedness* of terms.

In Nuprl the well-formedness of a term is stated using a ternary equality. The general term for the equality is $t_1 = t_2 \in T$. The semantic meaning of an open equality is that:

1. T is a type,
2. t_1 and t_2 are well-formed elements of type T,
3. and t_1 and t_2 are equal according to the equality of type T.

Every type provides an equality. It is possible to define another equality and thereby change a type using a *quotient* construction. Otherwise, a default equality is provided by the system. Membership $t \in T$ is stated as $t = t \in T$, which has the meaning that t is well-formed, and it is an element of type T. A peculiar characteristic of the equality is that the term $t_1 = t_2 \in T$ is itself not well-formed unless the first two properties of the equality hold; that is, T must be a type, and t_1 and t_2 must be elements of type T. Currently, there is no way to express non-membership.

Terms

The terms in Nuprl are declared using the *uniform representation* (Section 5.1) that makes binding and subterms explicit. The general form for any term in Nuprl is

$$opname\,[p_1{:}s_1, \ldots, p_n{:}s_n]\,\{\overline{v_1}.t_1; \cdots; \overline{v_m}.t_m\}\,.$$

Each term has an *operator* that is composed of an operator name, and a list of parameters, which are just pairs of strings. The operator is chosen so as to identify the term in a semantically meaningful way. For instance, the uniform representation of the type $\mathbb{Z}$ is the term *int* [] {}, and the uniform representation of the number 3 is the term *number* [$:3] {}. In most cases, the operator name identifies the major category of the operator, and the parameters subdivide the category. This is an implementation decision, so for instance, the numbers could be represented by the terms *number0* [] {} , *number1* [] {} , . . ., or by the term *number* [0] {} , *number* [1] {} , ... The latter representation

makes the interesting parts of the term more explicit, and is the preferred representation.

Each term also contains a list of bound subterms. In each subterm $\overline{v_i}.b_i$, there is a term b_i and a list of binding variables $\overline{v_i}$ that are bound in b_i. The uniform representation makes no claim as to the meaning of the binding; it is purely syntactical. As an example, the λ abstraction $\lambda v.b$ uses the uniform representation $lambda[]\{v.b\}$, and the definition of λ specifies the meaning of the components of the term.

Computation in Nuprl

The computation rules expressed for each type are of the form $t \to t'$, where t and t' are terms containing free variables. These computation rules can be used to express equivalences of terms. For example in any context $(\lambda v.v + 1)(2) \equiv 4$. The computational equivalences hold even if the terms do not have a well-defined type: for instance $(\lambda v.\text{“}hello'' + v)(\text{“}world'') \equiv \text{“}hello'' + \text{“}world''$, even though the terms themselves have no meaning.

The type theory includes evaluation of terms. If the evaluator terminates, it produces a canonical value equivalent to its argument. The evaluation is *leftmost–outermost*; it is a *lazy* evaluator. Of course, other evaluation orders are possible, and they may be preferred in some contexts. All evaluators are functional: if they halt, they provide equal values on equal arguments.

The canonical evaluation relation $\downarrow$ is defined as follows:

1. if t is a value, then $t \downarrow t$,
2. if $t \to t'$ and $t' \downarrow t''$, then $t \downarrow t''$,
3. let $s \equiv opname[\overline{p}]\{\overline{x_1}.t_1;\cdots;\overline{x_n}.t_n\}$ and let i be the smallest integer in the range $1 \le i \le n$ where
   (a) there are terms $t'_1,\ldots,t'_i$ such that $t_1 \downarrow t'_1,\ldots,t_i \downarrow t'_i$,
   (b) there is a computation rule $opname[\overline{p}]\left\{\overline{x_1}.t'_1;\cdots;\overline{x_i}.t'_i;\overline{x_{i+1}}.t_{i+1};\cdots;\overline{x_n}.t_n\right\} \to s'$,
   (c) there is an evaluation $s' \downarrow s''$,

   then $s \downarrow s''$.

The fact that a term can be typed is independent of whether the term terminates. The simplest example is the type $\bigcap_{x\in Void} T$ (Section 14.21), which is a well-formed type for any term T. Every term in Nuprl belongs to this type, because under the assumption that *Void* is inhabited

anything is true. Even the term $\Omega \equiv (\lambda x.xx)\,(\lambda x.xx) \in \bigcap_{x \in Void} T$, although $\Omega \to \Omega \to \Omega \to \cdots$. However, there are types where membership implies termination—in fact, almost all types in Nuprl require that their members have values. If $i \in \mathbb{Z}$, then i is equivalent to some value that is an integer. The values in $\mathbb{Z}$ are just numbers, and the canonical evaluator can be used to compute the number for i. Similarly if $t \in \mathbb{U}_i$, then t must be equivalent to some value that is a type. The value for t must be some base type, or type construction.

Sequent semantics

Theorems in Nuprl are expressed in the form of *sequents*, which have *hypotheses* and *conclusions*. Let $x_1{:}T_1, \ldots, x_n{:}T_n$ be a vector of named types, and let $C_1, \ldots, C_m$ be a vector of types dependant on $x_1, \ldots, x_n$. Then $x_1{:}T_1, \ldots, x_n{:}T_n \vdash C_1, \ldots, C_n$ is a sequent. In Nuprl, the sequents are further restricted to have only a single conclusion, which aids in enforcing constructivity.

In the sequent $x_1{:}T_1, \ldots, x_n{:}T_n \vdash C$, the variables $x_1, \ldots, x_{i-1}$ are bound in T_i for any i where $1 \le i \le n$. All variables $x_1, \ldots, x_n$ are bound in C.

A closed Nuprl sequent $x_1{:}T_1, \ldots, x_n{:}T_n \vdash C$ is *true* if, and only if:

- For each i in the range $1 \le i \le n$, the term T_i is a type, and it is pointwise-functional over $x_1, \ldots, x_{i-1}$, where $x_1, \ldots, x_{i-1}$ inhabit $T_1, \ldots, T_{i-1}$.
- C is a type, it is pointwise-functional over inhabitants $x_1, \ldots, x_n$ of $T_1, \ldots, T_n$.

A hypothesis T_i is pointwise functional if 1) $T_1, \ldots, T_{i-1}$ are pointwise functional, and 2) given hypotheses $x_1{:}T_1, \ldots, x_{i-1}{:}T_{i-1}$ and $y_1{:}T_1, \ldots, y_{i-1}{:}T_{i-1}$, with $x_1 = y_1 \in T_1, \ldots, x_{i-1} = y_{i-1} \in T_{i-1}$, then $T_i[x_1, \ldots, x_{i-1}]$ and $T_i[y_1, \ldots, y_{i-1}]$ are equal types. A similar definition applies to the conclusion.

A theorem in Nuprl is a proof of a sequent. An unconditional theorem has no hypotheses. The semantics of pointwise-functionality enforce the mathematical meaning of the theorems. The terms in the sequent have semantical meaning as real mathematical objects. Without this restriction, theorems could only be assumed to state proof theoretic properties about terms, and little underlying mathematical significance could be implied. This mathematical significance has it price in the complexity of proofs, for in addition to proving that a sequent it true, the prover must also verify its functionality.

Rule definitions

The proof rules for Nuprl can be divided into five categories:

1. *Equality* rules are used to prove well-formedness, membership, and equality.
2. *Formation* rules are used to construct inhabitants of a type.
3. *Elimination* rules are used to define function invocation, a case analysis, or induction on an assumption.
4. *Structural* rules are used to manipulate sequents,
5. *Computation* rules are used to define the operational semantics of Nuprl programs.

A feature of the Nuprl type theory is the incremental construction of proofs and the programs that they correspond to. Since the Nuprl logic is constructive,[1] every proof corresponds to a program that is a *construction* of the theorem. For instance, a proof of $\mathbb{Z}$ is any integer. In the Nuprl theory the proof terms, called *extracts*, are hidden by the proof system so that users do not have to deal with them explicitly. Since the proof term can be constructed syntactically from the proof tree, each rule in the logic has an extract component that determines how to construct the proof term. Once a theorem is proved, the system can construct the proof term from the proof tree and the rule templates.

Formation rules are used extensively in the interactive construction. To the user, the application of a formation rule is a *refinement* of the goal (viewed as a proposition) into subgoals. The formation rule constructs the program inhabiting the proof. Formation rules are not strictly necessary: to prove a proposition, the user could simply supply an inhabiting program, and prove its membership using equality reasoning. However, this approach makes incremental construction of programs difficult at best. A main goal of Nuprl is to raise the level of programming to reasoning about specifications, after which the programs can be constructed using the propositions-as-types correspondence. Reasoning about specifications is accomplished as an interactive sequence of refinements, and formations rules play a central part. Of course, since formation rules are redundant, they can be derived from the equality definition.

Each rule has

[1] Constructivity is not a requirement of the consistency of the Nuprl type theory—it is a deep logical fact proved by Howe [HS94] that the axiom of excluded middle could be added with no loss of consistency.

1. a name by which the rule can be invoked,
2. arguments to the invocation,
3. the rule,
4. a template for constructing the extract.

14.2 Itt_squash module

The `Itt_squash` module defines a resource that can be used to help prove "squash" stability—that is, to recover the proof of a proposition given an assumption that it is true.

Sequents in the Nuprl type theory have two forms: one is the generic form $[ext]\ H \vdash T$, where *ext* is a variable. The variable specifies that the proof extract is needed for the computational content of the proof.

The other form is $[squash]\ H \vdash T$, where `squash` is a term defined in the `Base_trivial` module. The `squash` term specifies that the proof extract is *not* needed for its computational content.

Typically, `squash` sequents are used for well-formedness goals (the computational content of well-formedness is never used), but `squash` sequents are also used in cases where the computational content can be inferred. Equality proofs $a = b \in T$ are the canonical example: the computational content of $a = b \in T$ is *always* the term $\cdot$, and proofs of equalities can always be squashed because the content can be discovered later.

Certain rules are valid only for `squash` sequents; rules for using squashed assertions in the "set" type (in the `Itt_set`) module), for example. Also, it can be argued that it is consistent to use *classical* reasoning on `squash` goals without losing constructive content.

This module defines a generic resource `squash_resource` that can be used to recover computational content from a `squash` proof. The resource defines a tactic `squashT` that performs the following inference:

$$\begin{array}{l} [squash]\ H \vdash T \\ \text{—BY squashT} \\ [ext]\ H \vdash T \end{array}$$

Additions to the resource require two arguments: a *term* that matches the conclusion of the squashed goal, and a tactic that performs squash reduction on goals of that form.

14.2.1 Parents

include Base_theory

14.2.2 Resources

The squash resource represents data using a table that maps goal terms to the tactic that performs squash inference on goals of that form. The argument `term * tactic` specifies the matching term and the tactic.

resource `squash_resource`**(**(term * tactic)**;**
tactic**;**
squash_data**;**
unit**)**

14.2.3 Tactics

The `squashT` tactic is defined as a proof annotation that is passed down the proof. The tactic retrieves the annotation and applies it to the current goal.

14.3 Itt_equal module

The `Itt_equal` module defines type *universes*, *cumulativity* of type universes, and equality.

14.3.1 Parents

include Base_theory
include Itt_squash

14.3.2 Terms

The universe type $\mathbb{U}_i$ is a *type of types*. The individual type universe judgments are listed in the modules for each of the types. Furthermore, there is currently no elimination rule.

The *cumulativity*$[i, j]$ term is a primitive judgment that defines level *inclusion* (this is a builtin judgment in MetaPRL).

The $'t$ *Type* term is used to define the *type* judgment. A term T is a type if $[squash]\ H \vdash T$ *Type*. This judgement is *not* part of the

standard Nuprl type theory [Con92, Con98, Con00], nor is it part of Martin–Löf's type theory [ML84b, ML75]. The judgment can be defined in the Nuprl type theory as $(\lambda x.x) \in (T \to T)$. However, to make a complete derivation in Nuprl, we would require an *elimination* form that would infer $T \in \mathbb{U}_i$ for *some* universe $\mathbb{U}_i$ if T *type*. Since a level expression i is always universally quantified, a rule for this inference is currently inexpressible. We take the judgement $T\ type$ as primitive. The rules for the type judgment are given in the modules for each of the type constructors.

The semantic meaning of an open equality is that:

1. T is a type,
2. t_1 and t_2 are well-formed elements of type T.
3. and t_1 and t_2 are equal using the equality of type T.

declare `type` = Itt_equal!type{'a}
declare `univ` = Itt_equal!univ[i:l]
declare `equal` = Itt_equal!equal{'T; 'a; 'b}
declare `cumulativity` = Itt_equal!cumulativity[i:l, j:l]

14.3.3 Rules

Equality axiom

The *axiom* rule declares that if a program x has type T by assumption, then T is a type, and x is a member of T.

! **rule** `equalityAxiom` H J:
  [*ext*] H; x: T; $J[x] \vdash x \in T$
  = .

Equality is an equivalence relation

The next three rules specify that equality is an equivalence relation. The *reflexivity* rule differs from the standard definition: a program x has type T if it is equal to any other element of T.

! **rule** `equalityRef` H y:
  [*ext*] $H \vdash x = y \in T \longrightarrow$
  [*ext*] $H \vdash x \in T$
  = .
! **rule** `equalitySym` H:
  [*ext*] $H \vdash y = x \in T \longrightarrow$

[*ext*] $H \vdash x = y \in T$
= ·

! **rule** `equalityTrans` H z:
[*ext*] $H \vdash x = z \in T \longrightarrow$
[*ext*] $H \vdash z = y \in T \longrightarrow$
[*ext*] $H \vdash x = y \in T$
= ·

! **rule** `equalityFormation` H T:
a: **[**main**]** [*ext*] $H \vdash T \longrightarrow$
b: **[**main**]** [*ext*] $H \vdash T \longrightarrow$
[*ext*] $H \vdash \mathbb{U}$[i]
= ($a = b \in T$)

Well-formedness of equality

The next two rules describe well-formedness of the equality judgment. Equality is *intensional*: two equalities are equal iff all of their parts are equal.

! **rule** `equalityEquality` **{|** intro_resource [] **|}** H:
[wf**]** [·] $H \vdash T1 = T2 \in \mathbb{U}$[i] $\longrightarrow$
[wf**]** [·] $H \vdash a1 = a2 \in T1 \longrightarrow$
[wf**]** [·] $H \vdash b1 = b2 \in T2 \longrightarrow$
[*ext*] $H \vdash a1 = b1 \in T1 = a2 = b2 \in T2 \in \mathbb{U}$[i]
= ·

! **rule** `equalityType` **{|** intro_resource [] **|}** H:
[wf**]** [·] $H \vdash a \in T \longrightarrow$
[wf**]** [·] $H \vdash b \in T \longrightarrow$
[*ext*] $H \vdash (a = b \in T)$ Type
= ·

Equality in a type

Equality in any term T means that T is a type.

! **rule** `equalityTypeIsType` H a b:
[wf**]** [·] $H \vdash a = b \in T \longrightarrow$
[*ext*] $H \vdash T$ Type
= ·

Inhabitants of the equality type

The two following rules state that · is the one-and-only element in a provable equality type.

! **rule** `axiomMember` **{|** intro_resource []**;** eqcd_resource **|}** H:
[wf**]** [·] $H \vdash a = b \in T \longrightarrow$
[*ext*] $H \vdash \cdot \in a = b \in T$
$= \cdot$

! **rule** `equalityElimination` **{|** elim_resource [] **|}** H J:
t: [*ext*] H; x: $a = b \in T$; $J[\cdot] \vdash C[\cdot] \longrightarrow$
[*ext*] H; x: $a = b \in T$; $J[x] \vdash C[x]$
$= t$

Truth implies typehood

For any sequent judgment [*ext*] $H \vdash T$ the term T must be a a type. The following rule allows us to infer well-formedness of a type from it provability. Note that this rule is useless for types T that are not true.

! **rule** `typeEquality` H:
[main**]** [·] $H \vdash T \longrightarrow$
[*ext*] $H \vdash T$ Type
$= \cdot$

Forgetting proof extracts

The following four rules state that the proof extract (the program that inhabits the equality type) can always be found, even if the proof is not known. This is always true because the equality type has only one proof (·).

! **rule** `equality_squashElimination` H:
[·] $H \vdash a = b \in T \longrightarrow$
[*ext*] $H \vdash a = b \in T$
$= \cdot$

! **rule** `type_squashElimination` H:
[·] $H \vdash T$ Type $\longrightarrow$
[*ext*] $H \vdash T$ Type
$= \cdot$

! **rule** `rewrite_squashElimination` H:
[·] $H \vdash a \longleftrightarrow b \longrightarrow$
[*ext*] $H \vdash a \longleftrightarrow b$
$= \cdot$

! **rule** `squashFromAny` H *ext*:
[*ext*] $H \vdash T \longrightarrow$
[·] $H \vdash T$
$= \cdot$

Universe cumulativity

The following two rules describe universe *cumulativity*. The *cumulativity*$[i : l, j : l]$ term is a built-in judgment describing level inclusion.

! **rule** `universeMember` H:
  [*ext*] $H \vdash$ cumulativity[j;i] $\longrightarrow$
  [*ext*] $H \vdash \mathbb{U}$[j] $\in \mathbb{U}$[i]
  $= \cdot$
! **rule** `universeCumulativity` H $\mathbb{U}$[j]:
  [$\cdot$] $H \vdash$ cumulativity[j;i] $\longrightarrow$
  [$\cdot$] $H \vdash x = y \in \mathbb{U}$[j] $\longrightarrow$
  [*ext*] $H \vdash x = y \in \mathbb{U}$[i]
  $= \cdot$

The type universe is a type

The next three rules state that every universe $\mathbb{U}_l$ is a type, and every inhabitant $x \in \mathbb{U}_l$ is also a type.

! **rule** `universeType` **{|** intro_resource [] **|}** H:
  [*ext*] $H \vdash \mathbb{U}$[l] Type
  $= \cdot$
! **rule** `universeMemberType` H $\mathbb{U}$[i]:
  [wf**]** [$\cdot$] $H \vdash x \in \mathbb{U}$[i] $\longrightarrow$
  [*ext*] $H \vdash x$ Type
  $= \cdot$
! **rule** `universeAssumType` H J:
  [*ext*] H; x: $\mathbb{U}$[l]; $J[x] \vdash x$ Type
  $= \cdot$

14.4 Itt_struct module

The `Itt_struct` module defines *structural* rules. Structural rules are logical operations like thinning and substitution that are not associated with a particular type.

14.4.1 Parents

include Itt_equal

14.4.2 Terms

The `bind` term defines the syntax for a binding term. It has no semantic meaning in the type theory.

declare `bind` = Itt_struct!bind{x. T['x]}

14.4.3 Rules

Axiom

The `hypothesis` rule defines proof by assumption: if A is assumed, then it is true. The proof extract term is the program denoted by the assumption $x{:}A$.

! **rule** `hypothesis` H J x:
  [*ext*] H; x: A; $J[x] \vdash A$
  = x

Thinning (of hypotheses)

The `thin` rule states that if the conclusion C can be proved from hypotheses defined in H and J, then it can also be proved with an additional assumption $x{:}A$. The name comes from the goal-directed view: the hypothesis $x{:}A$ is removed ("thinned") by the application of the rule.

Note that x must be a variable that is not bound by any hypothesis in H. The rule is even more stringent: x may not occur free in J or C (note that the goal is *not* phrased as $[ext]\ H; x{:}A; J[x] \vdash C$).

The proof extract term t is unchanged.

! **rule** `thin` H J:
  t: [*ext*] H; $J \vdash C \longrightarrow$
  [*ext*] H; x: A; $J \vdash C$
  = t

Cut (lemma instantiation)

The `cut` rule is an alternate form of *modus-ponens*. If the lemma S can be proved from the current assumptions H and J, and the goal T can be proved with this additional assumption, the lemma can be instantiated to obtain a proof of the goal.

The extract term is formed by instantiating the proof a of the lemma in the abstracted proof $f[x]$, to get a proof $f[a]$ of T.

! **rule** `cut` H J *S* x:
a: **[**assertion**]** [*ext*] *H*; *J*⊢ *S* ⟶
f[*x*]: **[**main**]** [*ext*] *H*; x: *S*; *J*⊢ *T* ⟶
[*ext*] *H*; *J*⊢ *T*
= *f*[*a*]

Explicit proof introduction

The `introduction` rule performs proof by explicit introduction of a proof term. If the program t has type T, then T is provable with proof extract t.

! **rule** `introduction` H *t*:
[wf**]** [·] *H*⊢ *t* ∈ *T* ⟶
[*ext*] *H*⊢ *T*
= *t*

Substitution

There are three rules to define substitution. The `substitution` rule defines substitution of an arbitrary subterm of the conclusion $T_{11}[t_1]$ with a new term t_2. For the substitution to be valid, the terms t_{11} and t_2 must be equal in some type T_2, the goal $T_1[t_2]$ must be provable, and the conclusion $T_1[x]$ must also be *functional* for arbitrary terms $x \in T_2$. Functionality means that the proofs of $T_1[x]$ must be equal for all terms $x \in T_2$; the `type` judgment enforces this restriction. This restriction allows the proof extract term t to be copied from the proof of $T_1[t_2]$.

The bind(*x*.*T1*[*x*]) argument specifies the exact location of the subterm to be replaced.

! **rule** `substitution` H (*t1* = *t2* ∈ *T2*) bind(*x*.*T1*[*x*]):
[equality**]** [·] *H*⊢ *t1* = *t2* ∈ *T2* ⟶
t: **[**main**]** [*ext*] *H*⊢ *T1*[*t2*] ⟶
[wf**]** [·] *H*; x: *T2*⊢ *T1*[*x*] Type ⟶
[*ext*] *H*⊢ *T1*[*t1*]
= *t*

Hypothesis substition is defined with two rules. The `hypReplacement` performs entire replacement of a hypothesis *A* with another *B*. The two types must be equal (in some universe). The proof extract is unchanged.

The `hypSubstitution` rule performs replacement of an arbitrary subterm in a hypothesis, in a similar manner to conclusion substitution.

! **rule** hypReplacement H J *B* 𝕌[i]:
t: **[**main**]** [*ext*] *H*; x: *B*; *J*[*x*]⊢ *T*[*x*] ⟶
[equality**]** [·] *H*; x: *A*; *J*[*x*]⊢ *A* = *B* ∈ 𝕌[i] ⟶
[*ext*] *H*; x: *A*; *J*[*x*]⊢ *T*[*x*]
= *t*
! **rule** hypSubstitution H J (*t1* = *t2* ∈ *T2*) bind(*y*.*A*[*y*]) z:
[equality**]** [·] *H*; x: *A*[*t1*]; *J*[*x*]⊢ *t1* = *t2* ∈ *T2* ⟶
t: **[**main**]** [*ext*] *H*; x: *A*[*t2*]; *J*[*x*]⊢ *T1*[*x*] ⟶
[wf**]** [·] *H*; x: *A*[*t1*]; *J*[*x*]; z: *T2*⊢ *A*[*z*] Type ⟶
[*ext*] *H*; x: *A*[*t1*]; *J*[*x*]⊢ *T1*[*x*]
= *t*

14.4.4 Tactics

Axiom

The `nthHypT` tactic uses the `hypothesis` rule to perform proof by assumption.

```
.
—BY nthHypTi
[ext] H; i: x: A; J ⊢ A
```

Thinning

The `thinT` tactic uses the `thin` rule to *thin* a hypothesis in the current goal.

```
[ext] H; J ⊢ C
—BY thinTi
[ext] H; i: x: A; J ⊢ C
```

The `thinAllT` tactic thins a sequence of hypothesis.

```
[ext] H; J ⊢ C
—BY thinAllTi j
[ext] H; i: x_i: A_i; · · · ; j: x_j: A_j; J ⊢ C
```

Lemma assertion

The `assertT` tactic uses the `cut` rule to introduce a lemma.

$$\begin{array}{l} \ldots \mathit{assertion} \ldots [\mathit{ext}]\ H \vdash A \\ [\mathit{ext}]\ H; x\colon A \vdash C \\ \text{—BY } \texttt{assertT}\, A \\ [\mathit{ext}]\ H \vdash C \end{array}$$

The `assertAtT` introduces the lemma at a specific location in the hypothesis list.

$$\begin{array}{l} \ldots \mathit{assertion} \ldots [\mathit{ext}]\ H; J \vdash A \\ [\mathit{ext}]\ H; x\colon A; J \vdash C \\ \text{—BY } \texttt{assertT}\, i\ A \\ [\mathit{ext}]\ H; (\mathit{location} i); J \vdash C \end{array}$$

Explicit witness introduction

The `useWitnessT` tactic uses the `introduction` rule to perform explicit proof witness introduction.

$$\begin{array}{l} [\mathit{squash}]\ H \vdash t \in T \\ \text{—BY } \texttt{useWitnessT}\, t \\ [\mathit{ext}]\ H \vdash T \end{array}$$

Substitution

The three substitution rules are unified into a single tactic `substT`, which takes a clause number i, and an equality $t_1 = t_2 \in T$. The tactic substitutes t_2 for *all* occurrences of the term t_1 in the clause. The following example illustrates the use.

$$\begin{array}{l} \ldots \mathit{equality} \ldots [\mathit{squash}]\ H \vdash 1 + 2 = 3 \in \mathbb{Z} \\ \ldots \mathit{main} \ldots [\mathit{ext}]\ H \vdash 3 < 1 * 3 \\ \ldots \mathit{wf} \ldots [\mathit{squash}]\ H; i\colon \mathbb{Z} \vdash (x < 1 * x)\ \mathit{Type} \\ \text{—BY } \texttt{substT}\, 1 + 2 = 3\, \mathit{in}\, \mathbb{Z} \\ [\mathit{ext}]\ H \vdash (1 + 2) < 1 * (1 + 2) \end{array}$$

14.4.5 Resources

The (`onSomeHypT nthHypT`) tactic is added to the `trivialT` resource.

14.5 Itt_subtype module

The `Itt_subtype` module provides the definition of *subtyping*. Informally a type T_1 is a subtype of a type T_2, $T_1 \subseteq T_2$, if any two equal elements of T_1 are also equal in T_2. This is slightly different from the set-theoretic definition. In set theory, the quotiented set $\mathbb{Z}_2$ contains two equivalence classes; one is the set of even numbers and the other is the set of odd numbers. In the Nuprl type theory, the two types have the same elements, but all even numbers are equal in $\mathbb{Z}_2$ (and all the odd numbers are also equal). It follows that $\mathbb{Z} \subseteq \mathbb{Z}_2$.

The subtype-type has trivial computational content, like equality. The subtype contains only the $\cdot$ term if it is true; it is empty otherwise.

14.5.1 Parents

include Itt_equal
include Itt_struct

14.5.2 Terms

The `subtype` term is a binary relation.

declare `subtype` = Itt_subtype!subtype{'A; 'B}

14.5.3 Rules

Typehood and equality

The $(A \subseteq B)$ term is a type if both A and B are types. The equality is *intensional*: two subtype-types are equal if their subterms are equal.

! **rule** `subtypeEquality` **{|** intro_resource []**;** eqcd_resource **|}** H:
[wf**]** [$\cdot$] $H \vdash$ *A1* = *A2* $\in \mathbb{U}$[i] $\longrightarrow$
[wf**]** [$\cdot$] $H \vdash$ *B1* = *B2* $\in \mathbb{U}$[i] $\longrightarrow$
[*ext*] $H \vdash$ (*A1* $\subseteq$ *B1*) = (*A2* $\subseteq$ *B2*) $\in \mathbb{U}$[i]
= $\cdot$

! **rule** `subtypeType` **{|** intro_resource [] **|}** H:
[wf**]** [$\cdot$] $H \vdash$ *A* Type $\longrightarrow$
[wf**]** [$\cdot$] $H \vdash$ *B* Type $\longrightarrow$
[*ext*] $H \vdash$ ($A \subseteq B$) Type
= $\cdot$

The intensional interpretation of typehood also means that if the subtype judgment $A \subseteq B$ is true, then both A and B are types.

! **rule** `subtypeTypeLeft` H A:
[main**]** [$\cdot$] $H \vdash A \subseteq B \longrightarrow$
[*ext*] $H \vdash B$ Type
$= \cdot$

! **rule** `subtypeTypeRight` H B:
[main**]** [$\cdot$] $H \vdash A \subseteq B \longrightarrow$
[*ext*] $H \vdash A$ Type
$= \cdot$

Introduction

The `subtype_axiomFormation` rule gives the introduction form for the subtype judgment. A type $A \subseteq B$ is true if A and B are types, and any term $t \in A$ is also in B. The proof extract term is always the $\cdot$ term.

! **rule** `subtype_axiomFormation` **{|** intro_resource [] **|}** H x:
[wf**]** [$\cdot$] $H \vdash A$ Type $\longrightarrow$
[main**]** [$\cdot$] H; x: $A \vdash x \in B \longrightarrow$
[*ext*] $H \vdash A \subseteq B$
$= \cdot$

Member equality

The subtype-type, if true, contains only the term $\cdot$. For $\cdot$ to be in $A \subseteq B$, the subtype judgment must be true.

! **rule** `subtype_axiomEquality`
{| intro_resource []**;** eqcd_resource **|}** H:
[main**]** [$\cdot$] $H \vdash A \subseteq B \longrightarrow$
[*ext*] $H \vdash \cdot \in (A \subseteq B)$
$= \cdot$

Elimination

Subtype elimination has two forms. The standard `subtypeElimination` form corresponds to induction: the witness $x{:}A \subseteq B$ is replaced with the unique proof term $\cdot$. The second rule `subtypeElimination2` takes a witness $a \in A$ and adds the additional assumption $a \in B$.

! **rule** subtypeElimination
{| elim_resource [(ThinOption thinT)] **|}** H J:
t**:** [*ext*] H; x: $A \subseteq B$; $J[\cdot] \vdash C[\cdot] \longrightarrow$
[*ext*] H; x: $A \subseteq B$; $J[x] \vdash C[x]$
$= t$
! **rule** subtypeElimination2 H J a y:
[wf**]** $[\cdot]$ H; x: $A \subseteq B$; $J[x] \vdash a \in A \longrightarrow$
$t[y]$**:** [*ext*] H; x: $A \subseteq B$; $J[x]$; y: $a \in B \vdash C[x] \longrightarrow$
[*ext*] H; x: $A \subseteq B$; $J[x] \vdash C[x]$
$= t[\cdot]$

Squash

The subtype_squashElimination rule allows the proof of the subtype to be omitted; the proof $\cdot$ can always be recovered.

! **rule** subtype_squashElimination H:
$[\cdot]$ $H \vdash A \subseteq B \longrightarrow$
[*ext*] $H \vdash A \subseteq B$
$= \cdot$

14.5.4 Resources

The Itt_subtype module defines the subtype_resource, which is used to prove subtyping judgments. The sub_resource_info argument requires two terms $t_1 \subseteq t_2$ that match the goal term, and a tactic that can be used to refine goals of that form.

resource sub_resource**(**sub_resource_info**;** tactic**;** sub_data**;** unit**)**

14.6 Itt_void module

The Itt_void module defines the *empty* type. The *Void* type is a subtype of every other type (since it has no elements).

14.6.1 Parents

include Itt_equal
include Itt_subtype

14.6.2 Rules

Equality and typehood

The *Void* type is a member of very universe, and it is a type.

! **rule** `voidEquality` {| intro_resource []; eqcd_resource |} H:
[*ext*] $H \vdash$ Void $\in \mathbb{U}$[i]
= ·

! **rule** `voidType` {| intro_resource [] |} H:
[*ext*] $H \vdash$ Void Type
= ·

Elimination

Since the *Void* type is empty, induction over the *Void* type produces no cases.

! **rule** `voidElimination` {| elim_resource [] |} H J:
[*ext*] H; x: Void; $J[x] \vdash C[x]$
= ·

Squash

Any proof of void can be squashed because the proof will never be used. This rule is added to the `squashT` resource.

! **rule** `void_squashElimination` H:
[·] $H \vdash$ Void $\longrightarrow$
[*ext*] $H \vdash$ Void
= ·

Subtyping

The *Void* type is a subtype of every other type. This rule is derived from the definition of subtyping, and the `voidElimination` rule.

[2,13]* **rule `void_subtype` H:
[*ext*] $H \vdash$ Void $\subseteq T$

14.7 Itt_unit module

The `Itt_unit` module defines a term containing exactly one element, ·. The element is the same term that inhabits the equality (Section 14.3) and subtype (Section 14.5) judgments.

14.7.1 Parents

include Itt_equal
include Itt_struct

14.7.2 Rules

Typehood and equality

The *Unit* type is a member of every universe, and it is also a type.

! **rule** `unitEquality` **{|** intro_resource []**;** eqcd_resource **|}** H:
[*ext*] $H \vdash$ Unit $\in \mathbb{U}$[i]
= ·

! **rule** `unitType` **{|** intro_resource [] **|}** H:
[*ext*] $H \vdash$ Unit Type
= ·

Introduction

The *Unit* type is always provable. The proof is the unique term ·.

! **rule** `unit_memberFormation` **{|** intro_resource [] **|}** H:
[*ext*] $H \vdash$ Unit
= ·

Membership

The unique inhabitant of the *Unit* type is the term ·.

! **rule** `unit_memberEquality`
{| intro_resource []**;** eqcd_resource **|}** H:
[*ext*] $H \vdash \cdot \in$ Unit
= ·

Elimination

The elimination rule `unitElimination` performs a case analysis on x: *Unit*. The witness is replaced with the term ·.

! **rule** `unitElimination`
{| elim_resource [(ThinOption thinT)] **|}** H J:
t: [*ext*] H; x: Unit; $J[\cdot] \vdash C[\cdot] \longrightarrow$
[*ext*] H; x: Unit; $J[x] \vdash C[x]$
$= t$

Squash

The proof extract for a proof on *Unit* can always be omitted; the proof is always the term $\cdot$. This rule is added to the `squashT` resource.

! **rule** `unit_squashElimination` H:
[$\cdot$] $H \vdash$ Unit $\longrightarrow$
[*ext*] $H \vdash$ Unit
$= \cdot$

Rewriting

Two terms in *Unit* are always computationally equivalent.
This rule is added to the `dT` resource for goals that match $t \longleftrightarrow Unit$.

**rule** `unitSqequal` H:
[$\cdot$] $H \vdash x = y \in$ Unit $\longrightarrow$
[*ext*] $H \vdash x \longleftrightarrow y$

14.8 Itt_atom module

The `Itt_atom` module defines the *Atom* type—a type of strings without any order relation. The elements of the atom type are the *tokens*. The only comparison of tokens is equality; there is no elimination rule.

The *Atom* type is defined as primitive. This is not strictly necessary; the type can be derived from the recursive type (Section 14.27).

14.8.1 Parents

include Itt_equal

14.8.2 Terms

The `atom` term defines the *Atom* type. The `token` term has a "token" parameter (a string) that defines the token. The display representation of a token $token(tok)$ is the quoted form "tok".

declare `atom` = Itt_atom!atom
declare `token` = Itt_atom!token[t:t]

14.8.3 Rules

Equality and typehood

The *Atom* term is a member of every universe, and it is a type.

! **rule** `atomEquality` **{|** intro_resource []**;** eqcd_resource **|}** H:
[*ext*] $H \vdash$ Atom $\in \mathbb{U}$[i]
= ·
! **rule** `atomType` **{|** intro_resource [] **|}** H:
[*ext*] $H \vdash$ Atom Type
= ·

Introduction

The *Atom* type is always provable; the token "t" is a witness.

! **rule** `tokenFormation` H 't':
[*ext*] $H \vdash$ Atom
= 't'

Membership

Two tokens are equal in the token type only if they are exactly the same token.

! **rule** `tokenEquality` **{|** intro_resource []**;** eqcd_resource **|}** H:
[*ext*] $H \vdash$ 't' $\in$ Atom
= ·

Two tokens in *Atom* are computationally equivalent if they are equal.

! **rule** `atomSqequal` H:
[·] $H \vdash x = y \in$ Atom $\longrightarrow$
[*ext*] $H \vdash x \longleftrightarrow y$
= ·

14.9 Itt_set module

The `Itt_set` module defines a "set" type, or more precisely, it defines a type by quantified *separation*. The form of the type is $\{x{:}T \mid P[x]\}$, where T is a type, and $P[x]$ is a type for any element $x \in T$. The elements of the set type are those elements of $x \in T$ where the proposition $P[x]$ is true.

The set type is a "squash" type: the type is similar to the dependent product $x{:}A \times P[x]$ (Section 14.15), but the proof $P[x]$ is omitted (squashed). The set type $\{x{:}T \mid P[x]\}$ is always a subtype of A.

14.9.1 Parents

include Itt_squash
include Itt_equal
include Itt_unit
include Itt_subtype
include Itt_struct

14.9.2 Terms

The `set` term defines the set type. The `hide` term is used to represent sequent hypotheses $B[x]$ that have been squashed by the set type (see the elimination form `setElimination`).

declare `set` = Itt_set!set{'A; x. B['x]}
declare `hide` = Itt_set!hide{'A}

14.9.3 Rules

Equality and typehood

The set type $\{x{:}A \mid B[x]\}$ is a type if A is a type, and $B[x]$ is a type for any $x \in A$. Equality of the set type is *intensional*. Two set types are equal only if their parts are equal. An alternative formulation would be to allow sets $\{x{:}A_1 \mid B_1[x]\}$ and $\{x{:}A_2 \mid B_2[x]\}$ to be equal if $A_1 = A_2 \in \mathbb{U}_i$ and $B_1[x] \Leftrightarrow B_2[x]$ for any $x \in A_2$.

! **rule** `setEquality` **{|** intro_resource []**;** eqcd_resource **|}** H x:
[wf**]** [·] $H \vdash A1 = A2 \in \mathbb{U}$[i] $\longrightarrow$
[wf**]** [·] H; x: $A1 \vdash B1[x] = B2[x] \in \mathbb{U}$[i] $\longrightarrow$

[*ext*] $H\vdash$ { a1:$A1$ | $B1[a1]$} = { a2:$A2$ | $B2[a2]$} $\in \mathbb{U}$[i]
= ·

! **rule** `setType` **{|** intro_resource [] **|}** H x:
[wf**]** [·] $H\vdash A1$ Type ⟶
[wf**]** [·] H; x: $A1\vdash B1[x]$ Type ⟶
[*ext*] $H\vdash$ { a1:$A1$ | $B1[a1]$} Type
= ·

Introduction

A set type $\{x\colon A \mid B[x]\}$ is true if there is an element $a \in A$ where $B[x]$ is true.

! **rule** `setMemberFormation` **{|** intro_resource [] **|}** H a z:
[wf**]** [·] $H\vdash a \in A$ ⟶
[main**]** [*ext*] $H\vdash B[a]$ ⟶
[wf**]** [·] H; z: $A\vdash B[z]$ Type ⟶
[*ext*] $H\vdash$ { x:A | $B[x]$}
= a

Membership

Two terms a_1 and a_2 are equal in the set type $\{a\colon A \mid B[a]\}$ if they are equal in A and also $B[a_1]$ is true.

! **rule** `setMemberEquality`
{| intro_resource []**;** eqcd_resource **|}** H x:
[wf**]** [·] $H\vdash a1 = a2 \in A$ ⟶
[assertion**]** [·] $H\vdash B[a1]$ ⟶
[wf**]** [·] H; x: $A\vdash B[x]$ Type ⟶
[*ext*] $H\vdash a1 = a2 \in$ { a:A | $B[a]$}
= ·

Elimination

An assumption with a set type $u\colon\{x\colon A \mid B[x]\}$ asserts two facts: that $u \in A$ and $B[u]$. However, the proof of $B[u]$ is unavailable. The **[**$B[u]$**]** hypothesis states that $B[u]$ is true, but its proof is omitted. Hidden hypotheses may be unhidden in a squashed sequent, when the proof extract term is unnecessary.

! **rule** `setElimination` **{|** elim_resource [] **|}** H J u v:
t**:** [*ext*] H; u: A; v: **[**$B[u]$**]**; $J[u]\vdash T[u]$ ⟶
[*ext*] H; u: { x:A | $B[x]$}; $J[u]\vdash T[u]$
= t

Subtyping

The set type $\{x{:}A \mid B[x]\}$ is always a subtype of A if the set type is really a type. This rule is added to the `subtype_resource`.

! **rule** `set_subtype` **{|** intro_resource [] **|}** H:
$[\cdot]\ H \vdash \{\ \mathrm{a}{:}A \mid B[a]\}\ \mathrm{Type} \longrightarrow$
$[ext]\ H \vdash \{\ \mathrm{a}{:}A \mid B[a]\} \subseteq A$
$= \cdot$

Unhiding

The next two rules allow special cases for hidden hypotheses to be unhidden. The first rule, `unhideEqual`, allows equalities to be unhidden (because the proof can always be inferred), and the `unhideGoalEqual` rule allows hypotheses to be unhidden if an equality is being proved (for the same reason).

! **rule** `unhideEqual` H J u:
$t[u]$**:** $[ext]\ H;\ \mathrm{u}{:}\ x = y \in A;\ J[\cdot] \vdash C[\cdot] \longrightarrow$
$[ext]\ H;\ \mathrm{u}{:}$ **[**$x = y \in A$**];** $J[u] \vdash C[u]$
$= t[\cdot]$
! **rule** `unhideGoalEqual` H J u:
$[\cdot]\ H;\ \mathrm{u}{:}\ P;\ J[\cdot] \vdash x[\cdot] = y[\cdot] \in T[\cdot] \longrightarrow$
$[ext]\ H;\ \mathrm{u}{:}$ **[**P**];** $J[u] \vdash x[u] = y[u] \in T[u]$
$= \cdot$

14.10 Itt_union module

The union type $T_1 + T_2$ defines a union space containing the elements of both T_1 and T_2. The union is *disjoint*: the elements are *tagged* with the `inl` and `inr` tags as belonging to the "left" type T_1 or the "right" type T_2.

The union type is the first primitive type that can have more than one element. The tag makes the handedness of membership decidable, and the union type $Unit + Unit$ contains two elements: (inl it) and (inr it). The `Itt_bool` module uses this definition to define the Boolean values, where *false* is (inl it) and *true* is (inr it).

14.10.1 Parents

include Itt_void
include Itt_equal
include Itt_struct
include Itt_subtype

14.10.2 Terms

The `union` type is the binary union of two types A and B. The elements are (inl 'a) for $a \in A$ and (inr b) for $b \in B$. The `decide` term *decides* the handedness of the term $x \in A + B$.

declare `union` = Itt_union!union{'A; 'B}
declare `inl` = Itt_union!inl{'x}
declare `inr` = Itt_union!inr{'x}
declare `decide` = Itt_union!decide{'x; y. a['y]; z. b['z]}

14.10.3 Rewrites

The following two rules define the computational behavior of the `decide` term. There are two reductions, the `reduceDecideInl` rewrite describes reduction of `decide` on the `inl` term, and `reduceDecideInr` describes reduction on the `inr` term. The rewrites are added to the `reduceC` resource.

! **rewrite** `reduceDecideInl` :
match (inl x) with inl u -> $l[u]$ | inr v -> $r[v]$ $\longleftrightarrow$
$l[x]$

! **rewrite** `reduceDecideInr` :
match (inr x) with inl u -> $l[u]$ | inr v -> $r[v]$ $\longleftrightarrow$
$r[x]$

14.10.4 Rules

Typehood and equality

The equality of the `union` type is intensional; the union $A + B$ is a type if both A and B are types.

! **rule** `unionEquality` **{|** intro_resource []**;** eqcd_resource **|}** H:
[wf**]** [·] $H \vdash A1 = A2 \in \mathbb{U}$[i] ⟶
[wf**]** [·] $H \vdash B1 = B2 \in \mathbb{U}$[i] ⟶
[*ext*] $H \vdash (A1 + B1) = (A2 + B2) \in \mathbb{U}$[i]
= ·
! **rule** `unionType` **{|** intro_resource [] **|}** H:
[wf**]** [·] $H \vdash A$ Type ⟶
[wf**]** [·] $H \vdash B$ Type ⟶
[*ext*] $H \vdash (A + B)$ Type
= ·

Introduction

The union type $A + B$ is true if both A and B are types, and either 1) A is provable, or 2) B is provable. The following two rules are added to the `dT` tactic. The application uses the `selT` tactic to choose the handedness; the `inlFormation` rule is applied with the tactic `selT 1 (dT 0)` and the `inrFormation` is applied with `selT 2 (dT 0)`.

! **rule** `inlFormation` **{|** intro_resource [(SelectOption 1)] **|}** H:
a: **[**main**]** [*ext*] $H \vdash A$ ⟶
[wf**]** [·] $H \vdash B$ Type ⟶
[*ext*] $H \vdash A + B$
= (inl *a*)
! **rule** `inrFormation` **{|** intro_resource [(SelectOption 2)] **|}** H:
b: **[**main**]** [*ext*] $H \vdash B$ ⟶
[wf**]** [·] $H \vdash A$ Type ⟶
[*ext*] $H \vdash A + B$
= (inr *b*)

Membership

The following two rules define membership, $inl(a) \in A + B$ if $a \in A$ and $inr(b) \in A + B$ if $b \in B$. Both A and B must be types.

! **rule** `inlEquality` **{|** intro_resource []**;** eqcd_resource **|}** H:
[wf**]** [·] $H \vdash a1 = a2 \in A$ ⟶
[wf**]** [·] $H \vdash B$ Type ⟶
[*ext*] $H \vdash$ (inl *a1*) = (inl *a2*) $\in (A + B)$
= ·
! **rule** `inrEquality` **{|** intro_resource []**;** eqcd_resource **|}** H:
[wf**]** [·] $H \vdash b1 = b2 \in B$ ⟶
[wf**]** [·] $H \vdash A$ Type ⟶
[*ext*] $H \vdash$ (inr *b1*) = (inr *b2*) $\in (A + B)$

= ·

Elimination

The handedness of the union membership is *decidable*. The elimination rule performs a case analysis in the assumption $x: A + B$; the first for the `inl` case, and the second for the `inr`. The proof extract term is the `decide` combinator (which performs a decision on element membership).

! **rule** `unionElimination`
{| elim_resource [(ThinOption thinT)] **|}** H J x u v:
left[*u*]**:** **[**left**]** [*ext*] *H*; x: *A* + *B*; u: *A*; *J*[inl *u*]⊢ *T*[inl *u*] ⟶
right[*u*]**:** **[**right**]** [*ext*] *H*; x: *A* + *B*; v: *B*; *J*[inr *v*]⊢ *T*[inr *v*] ⟶
[*ext*] *H*; x: *A* + *B*; *J*[*x*]⊢ *T*[*x*]
= match *x* with inl *u* -> *left*[*u*] | inr *v* -> *right*[*v*]

Combinator equality

The `decide` term equality is true if there is *some* type $A + B$ for which all the subterms are equal.

! **rule** `decideEquality`
{| intro_resource []**;**
eqcd_resource **|}** H bind(*z*.*T*[*z*]) (*A* + *B*) u v w:
[wf**]** [·] *H*⊢ *e1* = *e2* ∈ (*A* + *B*) ⟶
[wf**]** [·]
1. *H*
2. u: *A*
3. w: *e1* = (inl *u*) ∈ (*A* + *B*)
⊢ *l1*[*u*] = *l2*[*u*] ∈ *T*[(inl *u*)] ⟶
[wf**]** [·]
1. *H*
2. v: *B*
3. w: *e1* = (inr *v*) ∈ (*A* + *B*)
⊢ *r1*[*v*] = *r2*[*v*] ∈ *T*[(inr *v*)] ⟶
[*ext*]
1. *H*
⊢
match *e1* with inl *u1* -> *l1*[*u1*] | inr *v1* -> *r1*[*v1*]
= match *e2* with inl *u2* -> *l2*[*u2*] | inr *v2* -> *r2*[*v2*] ∈ *T*[*e1*]
= ·

Subtyping

The union type $A_1 + A_2$ is a subtype of type $A_2 + B_2$ if $A_1 \subseteq A_2$ and $B_1 \subseteq B_2$.

! **rule** `unionSubtype` **{|** intro_resource [] **|}** H:
[·] $H \vdash A1 \subseteq A2 \longrightarrow$
[·] $H \vdash B1 \subseteq B2 \longrightarrow$
[*ext*] $H \vdash (A1 + B1) \subseteq (A2 + B2)$
$= \cdot$

14.11 Itt_rfun module

The `Itt_rfun` module defines the *very-dependent function type*. This is the root type for the function types for the dependent-function `Itt_dfun` and the nondependent-function `Itt_fun`.

A complete description of the semantics of the type is given in Section 10. The type can be described informally as follows. The syntax of the type is $\{f \mid x{:}A \to B[f, x]\}$. The term f represents the functions f that are members of the type, the term A is the type of the *domain*, and given any particular function f in the type, and any argument $x \in A$, the type of the function *value* is the type $B[f, x]$. Roughly speaking, a function g is in the type $\{f \mid x{:}A \to B[f, x]\}$ if $g(x)$ has type $B[g, x]$ for any element $x \in A$. In addition, the domain A must be well-founded with a partial order on A, and the type $B[f, x]$ must be well-formed if f is restricted to arguments smaller the x.

14.11.1 Parents

include Itt_equal
include Itt_void
include Itt_set
include Itt_struct

14.11.2 Terms

The `rfun` type defines the very-dependent function $\{f \mid x{:}A \to B\}$; the `fun` type is used to define dependent functions $A \to x.B[x]$ and nondependent functions $A \to B$.

The elements of the function types are the functions $\lambda x.b[x]$, and the induction combinator is the application $f\ a$. The `fix` term defines a *fixpoint*.

The `well_founded` terms are used to define the well-founded order on the domain; the definition is given with the well-founded rules below.

declare `fun` = Itt_rfun!fun{'A; x. B['x]}
declare `rfun` = Itt_rfun!rfun{'A; f, x. B['f; 'x]}
declare `lambda` = Itt_rfun!lambda{x. b['x]}
declare `apply` = Itt_rfun!apply{'f; 'a}
declare `well_founded` = Itt_rfun!well_founded{'A; x, y. R['x; 'y]}
declare `well_founded_assum`
= Itt_rfun!well_founded_assum{'A; a1, a2. R['a1; 'a2]; 'P}
declare `well_founded_prop` = Itt_rfun!well_founded_prop{'A}
declare `well_founded_apply` = Itt_rfun!well_founded_apply{'P; 'a}
declare `fix` = Itt_rfun!fix{f. b['f]}

14.11.3 Rewrites

The `reduce_beta` rewrite defines normal beta-reduction. The `reduce_fix` rewrite defines reduction on the fixpoint combinator. The `reduce_fix` rewrite can be derived by defining the Y-combinator $Y \equiv \lambda f.\lambda x.(f\ (x\ x))\ (f\ (x\ x))$ and defining $fix(x.b[x]) \equiv Y\ (\lambda x.b[x])$.

! **rewrite** `reduce_beta` : $((\lambda v.b[v])\ a) \longleftrightarrow b[a]$
! **rewrite** `reduce_fix` : $\text{fix}(f.b[f]) \longleftrightarrow b[\text{fix}(f.b[f])]$

Well-foundedness

The following three rules are used to define a well-founded order. The term `well_founded_prop` *well_founded_prop*(A) represents an arbitrary proposition (predicate) on A; the `well_founded_apply` *well_founded_apply*$(P; a)$ represents the application of the proposition P to a. The `well_founded_assum` term *Itt_comment*!*math_well_founded_assum*

asserts that predicate P holds on all elements of a by induction on the relation $R[a_1, a_2]$.

The reason this definition is so convoluted is that the definition of well-foundedness must be give *before* defining functions and application. The `Itt_well_founded` module provides simplified definitions of well-foundedness.

! **rule** `well_founded_assum_elim`
{| elim_resource [(ThinOption thinT)] **|}** H J *a* a3 u:
[main**]** [·]
1. *H*
2. p: WellFounded ∀*a2*:*A*.(∀*a1*:*A*. *R*[*a1*; *a2*] →*P*[*a1*])→*P*[*a2*]
3. *J*[*p*]
⊢ *a* ∈ *A* ⟶
[main**]** [·]
1. *H*
2. p: WellFounded ∀*a2*:*A*.(∀*a1*:*A*. *R*[*a1*; *a2*] →*P*[*a1*])→*P*[*a2*]
3. *J*[*p*]
4. a3: *A*
5. u: *R*[*a3*; *a*]
⊢ *P*[*a3*] ⟶
t[*u*]**: [**main**]** [·]
1. *H*
2. p:
WellFounded ∀*a2*:*A*.(∀*a1*:*A*. *R*[*a1*; *a2*] →*P*[*a1*])→*P*[*a2*]
3. *J*[*p*]
4. u: *P*[*a*]
⊢ *C*[*p*] ⟶
[*ext*]
1. *H*
2. p: WellFounded ∀*a2*:*A*.(∀*a1*:*A*. *R*[*a1*; *a2*] →*P*[*a1*])→*P*[*a2*]
3. *J*[*p*]
⊢ *C*[*p*]
= *t*[·]
! **rule** `well_founded` **{|** intro_resource [] **|}** H a1 a2 a3 u v p P:
[wf**]** [·] *H*⊢ *A* Type ⟶
[wf**]** [·] *H*; a1: *A*; a2: *A*⊢ *R*[*a1*; *a2*] Type ⟶
[main**]** [·] *H*; a1: *A*; u: *R*[*a1*; *a1*]⊢ Void ⟶
[main**]** [·] *H*; a1: *A*; a2: *A*; u: *R*[*a1*; *a2*]; v: *R*[*a2*; *a1*]⊢ Void ⟶
[main**]** [·]
1. *H*
2. a1: *A*
3. a2: *A*
4. a3: *A*
5. u: *R*[*a1*; *a2*]
6. v: *R*[*a2*; *a3*]
⊢ *R*[*a1*; *a3*] ⟶
[main**]** [·]
1. *H*
2. a1: *A*

3. P: WellFounded A → Prop
4. p: WellFounded ∀$a3$:A.(∀$a2$:A. $R[a2; a3]$ →$P[a2]$)→$P[a3]$
⊢ $P[a1]$ ⟶
[*ext*] H⊢ WellFounded a,b:A.$R[a; b]$
= ·

! **rule** `well_founded_apply_type` **{|** intro_resource [] **|}** H A:
[wf**]** [·] H⊢ P ∈ WellFounded A → Prop ⟶
[wf**]** [·] H⊢ A ∈ 𝕌[i] ⟶
[wf**]** [·] H⊢ a ∈ A ⟶
[*ext*] H⊢ $P[a]$ ∈ 𝕌[i]
= ·

Typehood and equality

The well-formedness of the very-dependent function requires that the domain type A be a type, the the domain be well-founded with some relation R, and that $B[f,x]$ be a type for any restricted function $\{f \mid y{:}Itt_comment!math_setA; x.R[z,y] \to B[f,y]\}$.

! **rule** `rfunctionEquality`
{| intro_resource []**;** eqcd_resource **|}** H (λa.λb.$R[a; b]$) g y z:
[wf**]** [·] H⊢ $A1 = A2$ ∈ 𝕌[i] ⟶
[wf**]** [·] H⊢ WellFounded a,b:$A1$.$R[a; b]$ ⟶
[wf**]** [·]
1. H
2. y: $A1$
3. g: **{** f1 | x1:{ z:$A1$ | $R[z; y]$} → $B1[f1; x1]$ }
⊢ $B1[g; y] = B2[g; y]$ ∈ 𝕌[i] ⟶
[*ext*]
1. H
⊢
{ f1 | (a1:$A1$ → $B1[f1; a1]$) }
= **{** f2 | (a2:$A2$ → $B2[f2; a2]$) } ∈ 𝕌[i]
= ·

! **rule** `rfunctionType`
{| intro_resource [] **|}** H (λa.λb.$R[a; b]$) g y z:
[wf**]** [·] H⊢ A Type ⟶
[wf**]** [·] H⊢ WellFounded a,b:A.$R[a; b]$ ⟶
[wf**]** [·]
1. H
2. y: A
3. g: **{** f | x:{ z:A | $R[z; y]$} → $B[f; x]$ }
⊢ $B[g; y]$ Type ⟶
[*ext*] H⊢ **{** f | (a:A → $B[f; a]$) } Type

= ·

Introduction

Viewed as a proposition, the very-dependent function type represents a *recursive* proposition. The function type $\{f \mid x{:}A \to B[f,x]\}$ must be well-formed with well-founded order R on A, and $B[f,x]$ must be true for each $B[g,y]$ for $y \in A$ and g a function in the restricted function space $\{f \mid x{:}\{A{:}z \mid R[z,y]\} \to B[f,x]\}$ (the induction hypothesis). The proof extract term contains a fixpoint, due to the induction. The fixpoint is guaranteed to terminate because the domain is well-founded.

! **rule** `rfunction_lambdaFormation`
{| intro_resource [] **|}** H (λ*a*.λ*b*.*R*[*a*; *b*]) g y z:
[wf**]** [·] *H*⊢ *A* Type ⟶
[wf**]** [·] *H*⊢ WellFounded *a*,*b*:*A*.*R*[*a*; *b*] ⟶
b[*g*; *y*]**:** [*ext*]
1. *H*
2. y: *A*
3. g: **{** f | x:{ z:*A* | *R*[*z*; *y*]} → *B*[*f*; *x*] }
⊢ *B*[*g*; *y*] ⟶
[*ext*] *H*⊢ **{** f | x:*A* → *B*[*f*; *x*] }
= (λ*y*.fix(*g*.*b*[*g*; *y*]))

Membership

The members of the function space are the `lambda` terms. The function space must be well-formed, and the body of the function must inhabit the range type *B* where the `lambda` function is substituted for the function argument.

! **rule** `rfunction_lambdaEquality`
{| intro_resource []**;** eqcd_resource **|}** H y:
[wf**]** [·] *H*⊢ **{** f | (x:*A* → *B*[*f*; x]) } Type ⟶
[wf**]** [·] *H*; y: *A*⊢ *b1*[*y*] = *b2*[*y*] ∈ *B*[(λ*x1*.*b1*[*x1*]); *y*] ⟶
[*ext*] *H*⊢ (λ*x1*.*b1*[*x1*]) = (λ*x2*.*b2*[*x2*]) ∈ **{** f | (x:*A* → *B*[*f*; x]) }
= ·

Extensional equality

The function space is one of the few types in the Nuprl type theory with an *extensional* equality. Normally, equality is intensional — it depends

on the syntactic structure of the the terms in the type. The function space allows equality of functions f_1 and f_2 if their *application* provides equal values on equal arguments. The functions and the function type must all be well-formed (and in fact, this implicitly requires that f_1 and f_2 both be `lambda` terms).

! **rule** `rfunctionExtensionality`
H **{** g1 | (x1:*A1* → *B1*[*g1*; *x1*]) } **{** g2 | (x2:*A2* → *B2*[*g2*; *x2*]) } y:
[wf**]** [·] *H*⊢ **{** g | (x:*A* → *B*[*g*; *x*]) } Type ⟶
[main**]** [·] *H*; y: *A*⊢ *f1 y* = *f2 y* ∈ *B*[*f1*; *x*] ⟶
[wf**]** [·] *H*⊢ *f1* ∈ **{** g1 | (x1:*A1* → *B1*[*g1*; *x1*]) } ⟶
[wf**]** [·] *H*⊢ *f2* ∈ **{** g2 | (x2:*A2* → *B2*[*g2*; *x2*]) } ⟶
[*ext*] *H*⊢ *f1* = *f2* ∈ **{** g | (x:*A* → *B*[*g*; *x*]) }
= ·

Elimination

The elimination form for a function type $f\colon\{g \mid x\colon A \to B[g, x]\}$ allows *instantiation* of the function on an argument a, to get a proof $B[f, a]$.

! **rule** `rfunctionElimination` **{|** elim_resource [] **|}** H J f *a* y v:
[wf**]** [·] *H*; f: **{** g | x:*A* → *B*[*g*; *x*] }; *J*[*f*]⊢ *a* ∈ *A* ⟶
t[*f*; *y*; *v*]**:** [*ext*]
1. *H*
2. f: **{** g | x:*A* → *B*[*g*; *x*] }
3. *J*[*f*]
4. y: *B*[*f*; *a*]
5. v: *y* = *f a* ∈ *B*[*f*; *a*]
⊢ *T*[*f*] ⟶

[*ext*] *H*; f: **{** g | x:*A* → *B*[*g*; *x*] }; *J*[*f*]⊢ *T*[*f*]
= *t*[*f*; (*f a*); ·]

Combinator equality

Two *applications* are equal if their functions are equal, and their arguments are equal.

! **rule** `rfunction_applyEquality`
{| eqcd_resource **|}** H **{** f | (x:*A* → *B*[*f*; *x*]) }:
[wf**]** [·] *H*⊢ *f1* = *f2* ∈ **{** f | (x:*A* → *B*[*f*; *x*]) } ⟶
[wf**]** [·] *H*⊢ *a1* = *a2* ∈ *A* ⟶
[*ext*] *H*⊢ *f1 a1* = *f2 a2* ∈ *B*[*f1*; *a1*]
= ·

Subtyping

Function subtyping is *contravariant* in the domain and *covariant* in the range. The range subtyping is given using an arbitrary instance f in the function space.

[15,94]* **rule `rfunction_rfunction_subtype`
{| intro_resource [] **|}** H a f ($\lambda a.\lambda b.R[a; b]$):
[main**]** [·] $H \vdash A2 \subseteq A1 \longrightarrow$
[wf**]** [·] $H \vdash$ **{** f1 | (x1:$A1 \to B1[f1; x1]$) } Type $\longrightarrow$
[wf**]** [·] $H \vdash$ **{** f2 | (x2:$A2 \to B2[f2; x2]$) } Type $\longrightarrow$
[wf**]** [·] H; a1: $A1$; a2: $A1 \vdash R[a; b]$ Type $\longrightarrow$
[main**]** [·]
1. H
2. f: **{** f1 | x1:$A1 \to B1[f1; x1]$ }
3. a: $A2$
$\vdash B1[f; a] \subseteq B2[f; a] \longrightarrow$

[*ext*]
1. H
$\vdash$ **{** f1 | (x1:$A1 \to B1[f1; x1]$) } $\subseteq$ **{** f2 | (x2:$A2 \to B2[f2; x2]$) }

14.12 Itt_dfun module

The `Itt_dfun` module is *derived* from the `Itt_rfun` module. The type $x{:}A \to B[x]$ is equivalent to the type $\{f \mid x{:}A \to B[x]\}$, where f is not bound in $B[x]$. The *well-founded* restriction in for the very-dependent function type is always trivially satisfied (since the range type $B[x]$ never invokes f). The `Itt_dfun` module derives the dependent-function rules.

14.12.1 Parents

include Itt_equal
include Itt_rfun
include Itt_struct
include Itt_void
include Itt_unit

14.12.2 Rewrites

The `unfold_dfun` gives the definition of the dependent-function space.

! **rewrite** `unfold_dfun` : (x:$A \to B[x]$) $\longleftrightarrow$ **{** f | (x:$A \to B[x]$) **}**

14.12.3 Rules

Lemmas

The `void_well_founded` rule is a lemma that is useful for proving the well-formedness of the dependent-function space. The `void` type is trivially well-founded, since it has no elements.

[3,60]* **rule `void_well_founded` **{|** intro_resource [] **|}** H:
[wf**]** [·] $H \vdash A$ Type $\longrightarrow$
[*ext*] $H \vdash$ WellFounded *a1*,*a2*:*A*.Void
[6,32]* **rule `functionEquality`
{| intro_resource []**;** eqcd_resource **|}** H x:
[wf**]** [·] $H \vdash A1 = A2 \in \mathbb{U}$[i] $\longrightarrow$
[wf**]** [·] H; x: $A1 \vdash B1[x] = B2[x] \in \mathbb{U}$[i] $\longrightarrow$
[*ext*] $H \vdash$ (a1:$A1 \to B1[a1]$) = (a2:$A2 \to B2[a2]$) $\in \mathbb{U}$[i]
[2,31]* **rule `functionType` **{|** intro_resource [] **|}** H x:
[wf**]** [·] $H \vdash A1$ Type $\longrightarrow$
[wf**]** [·] H; x: $A1 \vdash B1[x]$ Type $\longrightarrow$
[*ext*] $H \vdash$ (a1:$A1 \to B1[a1]$) Type

Introduction

The propositional interpretation of the dependent-function is the universal quantification `all`, $\forall a{:}A.B[x]$. The universal quantification is true, if it is a type, and $B[a]$ is true for any $a \in A$.

[2,31]* **rule `lambdaFormation` **{|** intro_resource [] **|}** H z:
[wf**]** [·] $H \vdash A$ Type $\longrightarrow$
$b[z]$**: [**main**]** [*ext*] H; z: $A \vdash B[z]$ $\longrightarrow$
[*ext*] $H \vdash$ a:$A \to B[a]$

Membership

The dependent function space contains the `lambda` functions.

[7,34]* **rule `lambdaEquality` **{|** intro_resource [] **|}** H x:
[wf**]** [·] $H \vdash A$ Type $\longrightarrow$
[wf**]** [·] H; x: $A \vdash b1[x] = b2[x] \in B[x]$ $\longrightarrow$
[*ext*] $H \vdash (\lambda a1.b1[a1]) = (\lambda a2.b2[a2]) \in$ (a:$A \to B[a]$)

Extensionality

The dependent-function retains the extensional membership equality of the very-dependent function type. This rule is derived from the `rfunctionExtensionality` rule.

[9,50]* **rule `functionExtensionality`
H (y:$C \to D[y]$) (z:$E \to F[z]$) u:
[main**]** [·] H; u: $A \vdash f\ u = g\ u \in B[u] \longrightarrow$
[wf**]** [·] $H \vdash A$ Type $\longrightarrow$
[wf**]** [·] $H \vdash f \in$ (y:$C \to D[y]$) $\longrightarrow$
[wf**]** [·] $H \vdash g \in$ (z:$E \to F[z]$) $\longrightarrow$
[*ext*] $H \vdash f = g \in$ (x:$A \to B[x]$)

Elimination

The elimination rule *instantiates* the function $f{:}x{:}A \to B[x]$ with an argument $a \in A$, to obtain a proof of $B[a]$.

[3,20]* **rule `functionElimination`
{| elim_resource [] **|}** H J f a y v:
[wf**]** [·] H; f: x:$A \to B[x]$; $J[f] \vdash a \in A \longrightarrow$
$t[f; y; v]$**:** [*ext*]
1. H
2. f: x:$A \to B[x]$
3. $J[f]$
4. y: $B[a]$
5. v: $y = f\ a \in B[a]$

$\vdash T[f] \longrightarrow$
[*ext*] H; f: x:$A \to B[x]$; $J[f] \vdash T[f]$

Combinator equality

Applications have (at least) an *intensional* equality; they are equal if their functions and arguments are equal.

[3,11]* **rule `applyEquality` **{|** eqcd_resource **|}** H (x:$A \to B[x]$):
[·] $H \vdash f1 = f2 \in$ (x:$A \to B[x]$) $\longrightarrow$
[·] $H \vdash a1 = a2 \in A \longrightarrow$
[*ext*] $H \vdash f1\ a1 = f2\ a2 \in B[a1]$

Subtyping

Function spaces are *contravariant* in the domains, and *covariant* in their ranges. More specifically, the ranges must be pointwise covariant.

[31,195]* **rule `functionSubtype` **{|** intro_resource [] **|}** H a:
[·] $H \vdash A2 \subseteq A1 \longrightarrow$
[·] H; a: $A1 \vdash B1[a] \subseteq B2[a] \longrightarrow$
[*prop*] $H \vdash$ (a1:$A1 \to B1[a1]$) $\subseteq$ (a2:$A2 \to B2[a2]$)

14.13 Itt_fun module

The `Itt_fun` module defines the non-dependent function type. The function type is *derived* from the dependent-function type `Itt_dfun`, which is in turn derived from the very-dependent function `Itt_rfun`.

The non-dependent function $A \to B$ is the type of functions with domain A, and range B. It is equivalent to the dependent function space $x{:}A \to B$, where x is not bound in B.

14.13.1 Parents

include Itt_equal
include Itt_dfun

14.13.2 Rewrites

The non-dependent function $A \to B$ is defined as the dependent function $x{:}A \to B$ (where x is new).

! **rewrite** `unfold_fun` : $(A \to B) \longleftrightarrow$ (x:$A \to B$)

Typehood and equality

The non-dependent function has an intensional type equality.

[3,14]* **rule `independentFunctionEquality`
{| intro_resource []**;** eqcd_resource **|}** H:
[wf**]** [·] $H \vdash A1 = A2 \in \mathbb{U}$[i] $\longrightarrow$
[wf**]** [·] $H \vdash B1 = B2 \in \mathbb{U}$[i] $\longrightarrow$
[*ext*] $H \vdash (A1 \to B1) = (A2 \to B2) \in \mathbb{U}$[i]

*[1,10] **rule** independentFunctionType {| intro_resource [] |} H x:
[wf**]** [·] *H*⊢ *A1* Type ⟶
[wf**]** [·] *H*; x: *A1*⊢ *B1* Type ⟶
[*ext*] *H*⊢ (*A1* → *B1*) Type

Introduction

The propositional interpretation of the function space $A \to B$ is the implication term `implies`, $A \Rightarrow B$. The proposition is true if it is a type, and for any proof of A, there is a proof of B.

*[1,10] **rule** independentLambdaFormation
{| intro_resource [] |} H z:
[wf**]** [·] *H*⊢ *A* Type ⟶
b[*z*]**: [**main**]** [*ext*] *H*; z: *A*⊢ *B* ⟶
[*ext*] *H*⊢ *A* → *B*

Membership

The elements in the function space $A \to B$ are the `lambda` functions. The space $A \to B$ must be a type, and the body of the function must be an B for any argument in A.

*[1,10] **rule** independentLambdaEquality
{| intro_resource []**;** eqcd_resource **|}** H x:
[wf**]** [·] *H*⊢ *A* Type ⟶
[wf**]** [·] *H*; x: *A*⊢ *b1*[*x*] = *b2*[*x*] ∈ *B* ⟶
[*ext*] *H*⊢ (λ*a1*.*b1*[*a1*]) = (λ*a2*.*b2*[*a2*]) ∈ (*A* → *B*)

Elimination

There are two elimination forms. The `independentFunctionElimination` rule is more appropriate for the propositional interpretation of the function space $A \to B$: if there is a proof of A, then there is also a proof of B. The second form, `independentFunctionElimination2`, is more appropriate for the functional application to a specific argument $a \in A$.

*[4,32] **rule** independentFunctionElimination H J f y:
*a***: [**assertion**]** [*ext*] *H*; f: *A* → *B*; *J*[*f*]⊢ *A* ⟶
t[*f*; *y*]**: [**main**]** [*ext*] *H*; f: *A* → *B*; *J*[*f*]; y: *B*⊢ *T*[*f*] ⟶
[*ext*] *H*; f: *A* → *B*; *J*[*f*]⊢ *T*[*f*]
*[3,20] **rule** independentFunctionElimination2 H J f y z *a*:
[wf**]** [·] *H*; f: *A* → *B*; *J*[*f*]⊢ *a* ∈ *A* ⟶

$t[y; z]$: [main] [*ext*]
1. H
2. f: $A \to B$
3. $J[f]$
4. y: B
5. z: $y = f\ a \in B$

$\vdash T[f] \longrightarrow$
[*ext*] H; f: $A \to B$; $J[f] \vdash T[f]$

Combinator equality

Applications have an intensional equality; they are equal if their functions and arguments are equal.

[3,11]* **rule `independentApplyEquality`
{| eqcd_resource |} H ($A \to B$):
[wf] [·] $H \vdash f1 = f2 \in (A \to B) \longrightarrow$
[wf] [·] $H \vdash a1 = a2 \in A \longrightarrow$
[*ext*] $H \vdash f1\ a1 = f2\ a2 \in B$

Subtyping

The function space is *contravariant* in their domains, and *covariant* in their ranges.

14.14 Itt_well_founded module

The `Itt_well_founded` module provides a more convenient description of well-foundedness than the `well_founded_prop` term formalized the the `Itt_rfun` module. The definition of well-foundedness requires the derivation of an induction principle.

14.14.1 Parents

include Itt_fun
include Itt_logic

14.14.2 Terms

The `partial_order` term specifies that R is a partial order on type A. The `well_founded` term specifies that, in addition, R is a well-founded order on A.

declare `partial_order`
= Itt_well_founded!partial_order{'A; x, y. R['x; 'y]}
declare `well_founded`
= Itt_well_founded!well_founded[i:l]{'A; x, y. R['x; 'y]}

14.14.3 Rewrites

The definition of `partial_order` is that the order is anti-reflexive, anti-symmetric, and transitive.

! **rewrite** `unfold_partial_order` :
partial_order(x,y:A. $R[x; y]$) $\longleftrightarrow$
$((\forall x{:}A.\ \neg R[x; x])$
$\wedge\ (\forall x{:}A.\ \forall y{:}A.\ (R[x; y] \rightarrow \neg R[y; x]))$
$\wedge\ (\forall x{:}A.\ \forall y{:}A.\ \forall z{:}A.\ (R[x; y] \rightarrow R[y; z] \rightarrow R[x; z]))$

The definition of `well_founded` requires that R be a partial order, and that there is an *induction* principle that can be used to prove any predicate P on A. This is different from the classical definition (that there are no infinite descending chains), but the induction principle implies that classical property.

! **rewrite** `unfold_well_founded` :
well_founded[i](x,y:A. $R[x; y]$) $\longleftrightarrow$
(partial_order(x,y:A. $R[x; y]$)
$\wedge\ (\forall P{:}(A \rightarrow \mathbb{U}[i])$
$((\forall x{:}A.\ ((\forall y{:}A.\ (R[y; x] \rightarrow P\ y)) \rightarrow P\ x)) \rightarrow (\forall x{:}A.\ P\ x))))$

14.14.4 Rules

Well-formedness

The `partial_order` and `well_founded` predicates are both well-formed if their domain A is a type, and their relation R is a binary relation.

*[1,542] **rule** partial_order_type {| intro_resource [] |} H a b:
[wf**]** [·] $H \vdash A$ Type $\longrightarrow$
[wf**]** [·] H; a: A; b: $A \vdash R[a; b]$ Type $\longrightarrow$
[*ext*] $H \vdash$ partial_order(x,y:A. $R[x; y]$) Type
*[1,231] **rule** well_founded_type {| intro_resource [] |} H a b:
[wf**]** [·] $H \vdash A$ Type $\longrightarrow$
[wf**]** [·] H; a: A; b: $A \vdash R[a; b]$ Type $\longrightarrow$
[*ext*] $H \vdash$ well_founded[i](x,y:A. $R[x; y]$) Type

The purpose of this definition is to give a more convenient specification of well-foundedness that uses normal quantification in its formalization (the well_founded_prop predicate defined in the Itt_rfun can't use the function type in its definition). The following rule specifies that the new description of well-foundedness is sufficient to derive the primitive definition.

*[33,830] **rule** well_founded_reduction H a b $\mathbb{U}$[i]:
[wf**]** [·] $H \vdash A \in \mathbb{U}$[i] $\longrightarrow$
[wf**]** [·] H; a: A; b: $A \vdash R[a; b] \in \mathbb{U}$[i] $\longrightarrow$
[main**]** [*ext*] $H \vdash$ well_founded[i](x,y:A. $R[x; y]$) $\longrightarrow$
[*ext*] $H \vdash$ WellFounded x,y:A.$R[x; y]$

14.15 Itt_dprod module

The *dependent product* x:$A \times B$ is a space of *pairs*, where A is a type, and $B[x]$ is a family of types indexed by $x \in A$. The elements of the product space are the pairs (a, b), where $a \in A$ and $b \in B[a]$.

The dependent product uses an *intensional* membership equality. The only elements are the pair terms, which allows a strong elimination rule productElimination of the form:

rule productElimination i =
[*ext*] $H; a$: $A; b$: $B[a]; J[(a, b)] \vdash C[(a, b)]$
[*ext*] $H; x$: y: $A \times B[u]; J[x] \vdash C[x]$.

Because of this intensional equality, the Itt_dprod module is primitive. An alternative formulation would be to *derive* the product space from the very-dependent function rfun (in the Itt_rfun module), the union (in the Itt_union module), and the unit type (in the and Itt_union) module). The construction is as follows.

First, the Boolean values are defined with the union and unit types.

$$\begin{array}{rcl} \mathit{Bool} & \equiv & \mathit{Unit} + \mathit{Unit} \\ \mathit{ff} & \equiv & \mathit{inl}(\cdot) \\ \mathit{tt} & \equiv & \mathit{inr}(\cdot) \\ \textbf{if}\, e_1\, \textbf{then}\, e_2\, \textbf{else}\, e_3 & \equiv & \textbf{match}\, e_1\, \textbf{with}\, \mathit{inl}(_) \rightarrow e_2 \mid \mathit{inr}(_) \rightarrow e_3 \end{array}$$

Next, the dependent product space is defined as a function on a Boolean domain.

$$x{:}A \times B[x] \;\equiv\; \{f \mid x{:}\mathit{Bool} \rightarrow \textbf{if}\, x\, \textbf{then}\, A\, \textbf{else}\, B[f(\mathit{ff})]\}$$

The elements of this type are the functions that return the first projection on the $\bot$ argument, and the second projection on the *tt* argument.

$$\begin{array}{rcl} (a, b) & \equiv & \lambda x.\textbf{if}\, x\, \textbf{then}\, a\, \textbf{else}\, b \\ \mathit{fst}(p) & \equiv & p(\bot) \\ \mathit{snd}(p) & \equiv & p(\mathit{tt}) \end{array}$$

This encoding is satisfactory in all respects except for the elimination form. The problem is that the function space uses an *extensional* equality; the elements of the function space $\{f \mid x{:}\mathit{Bool} \rightarrow \textbf{if}\, x\, \textbf{then}\, A\, \textbf{else}\, B[f(\bot)]\}$ are not just the terms $\lambda x.\textbf{if}\, x\, \textbf{then}\, a\, \textbf{else}\, b$, but all equal functions.

One alternative is to weaken the elimination form to a more extensional version:

rule weakProductElimination p =
$[\mathit{ext}]\ \ldots; a{:}A; b{:}B[a]; w{:}(a, b) = p \in x{:}A \times B[x] \vdash C[p]$
$[\mathit{ext}]\ H; p{:}x{:}A \times B[x]; J[p] \vdash C[p]$.

This rule *can* be derived from the very-dependent function encoding, but it is probably too weak to be useful. For this reason the `Itt_dprod` module and the `prod` type are defined as primitive.

14.15.1 Parents

include Itt_void
include Itt_equal
include Itt_struct

include Itt_subtype

14.15.2 Terms

The `prod` term defines the dependent-product space, and the `pair` term defines the elements. The `spread` term is the induction combinator that performs a pattern match on the argument *e*. The `fst` and `snd` terms are the normal projections derived from the `spread` term.

declare `prod` = Itt_dprod!prod{'A; x. B['x]}
declare `pair` = Itt_dprod!pair{'a; 'b}
declare `spread` = Itt_dprod!spread{'e; u, v. b['u; 'v]}
declare `fst` = Itt_dprod!fst{'e}
declare `snd` = Itt_dprod!snd{'e}

14.15.3 Rewrites

The `spread` term performs a pattern match and substitutes the components of the pair into the body term *c*. This reduction is added to the `reduceC` resource.

! **rewrite** `reduceSpread` :
(**match** (u,v) **with** $(a,b) \rightarrow c[a; b]) \longleftrightarrow$
$c[u; v]$

The `fst` and `snd` terms are simplified forms of `spread` that produce the first and second projections. They also have derived reduction rules that are also added to the `reduceC` resource.

! **rewrite** `unfoldFst` : $e.1 \longleftrightarrow$ (**match** e **with** $(u,v) \rightarrow u$)
! **rewrite** `unfoldSnd` : $e.2 \longleftrightarrow$ (**match** e **with** $(u,v) \rightarrow v$)
[1,20]* **rewrite `reduceFst` : $(a,b).1 \longleftrightarrow a$
[1,20]* **rewrite `reduceSnd` : $(a,b).2 \longleftrightarrow b$

14.15.4 Rules

Typehood and equality

The type equality is intensional. A product type $(x{:}A \times B[x])$ is a type if A is a type, and $B[x]$ is a family of types indexed by $x \in A$.

! **rule** `productEquality`
{| intro_resource []**;** eqcd_resource **|}** H y:
[wf**]** [·] $H \vdash A1 = A2 \in \mathbb{U}$[i] ⟶
[wf**]** [·] H; y: $A1 \vdash B1[y] = B2[y] \in \mathbb{U}$[i] ⟶
[*ext*] $H \vdash (x1{:}A1 \times B1[x1]) = (x2{:}A2 \times B2[x2]) \in \mathbb{U}$[i]
= ·

! **rule** `productType` **{|** intro_resource [] **|}** H x:
[wf**]** [·] $H \vdash A1$ Type ⟶
[wf**]** [·] H; x: $A1 \vdash A2[x]$ Type ⟶
[*ext*] $H \vdash (y{:}A1 \times A2[y])$ Type
= ·

Introduction

The propositional interpretation of the product space is the bounded existential $\exists x{:}A.B[x]$. The proposition is true if it is a type, and if there is some element $a \in A$ where $B[a]$ is true.

! **rule** `pairFormation` **{|** intro_resource [] **|}** H a y:
[wf**]** [·] $H \vdash a \in A$ ⟶
b**: [**main**]** [*ext*] $H \vdash B[a]$ ⟶
[wf**]** [·] H; y: $A \vdash B[y]$ Type ⟶
[*ext*] $H \vdash x{:}A \times B[x]$
= (a,b)

Membership

The elements of the product space are the pairs. For the equality judgment to be valid, the $x{:}A \times B[x]$ term must be a type. The first subgoal ensures that A is a type, and the second ensures that $B[a_1]$ is a type, but $B[y]$ must be well-formed for *any* element $y \in A$, which is the purpose of the third subgoal.

! **rule** `pairEquality` **{|** intro_resource []**;** eqcd_resource **|}** H y:
[wf**]** [·] $H \vdash a1 = a2 \in A$ ⟶
[wf**]** [·] $H \vdash b1 = b2 \in B[a1]$ ⟶
[wf**]** [·] H; y: $A \vdash B[y]$ Type ⟶
[*ext*] $H \vdash (a1,b1) = (a2,b2) \in (x{:}A \times B[x])$
= ·

Elimination

The elimination rule performs a case analysis on the hypothesis $x{:}x{:}A \times B[x]$. The *only* elements are the pairs, and the rule splits the

hypothesis into its parts. The proof extract term is the `spread` induction combinator.

! **rule** `productElimination`
{| elim_resource [(ThinOption thinT)] **|}** H J z u v:
$t[u; v]$: **[**wf**]** [*ext*]
1. H
2. z: x:$A \times B[x]$
3. u: A
4. v: $B[u]$
5. $J[(u,v)]$
$\vdash T[(u,v)] \longrightarrow$
[*ext*] H; z: x:$A \times B[x]$; $J[z] \vdash T[z]$
= (**match** z **with** $(u,v) \rightarrow t[u; v]$)

Combinator equality

The `spread` combinator is well-formed if all its parts are well-formed.

! **rule** `spreadEquality`
{| eqcd_resource **|}** H bind($z.T[z]$) (w:$A \times B[w]$) u v a:
[wf**]** [$\cdot$] $H \vdash e1 = e2 \in (w{:}A \times B[w]) \longrightarrow$
[wf**]** [$\cdot$]
1. H
2. u: A
3. v: $B[u]$
4. a: $e1 = (u,v) \in (w{:}A \times B[w])$
$\vdash b1[u; v] = b2[u; v] \in T[(u,v)] \longrightarrow$
[*ext*]
1. H
$\vdash$
(**match** $e1$ **with** $(u1,v1) \rightarrow b1[u1; v1]$)
= (**match** $e2$ **with** $(u2,v2) \rightarrow b2[u2; v2]) \in T[e1]$
= $\cdot$

Subtyping

The subtype judgment is *covariant* in the first type A, and pointwise-covariant in the second type $B[a]$ for each $a \in A$.

! **rule** `productSubtype` **{|** intro_resource [] **|}** H a:
[$\cdot$] $H \vdash A1 \subseteq A2 \longrightarrow$
[$\cdot$] H; a: $A1 \vdash B1[a] \subseteq B2[a] \longrightarrow$
[*ext*] $H \vdash (a1{:}A1 \times B1[a1]) \subseteq (a2{:}A2 \times B2[a2])$
= $\cdot$

14.16 Itt_prod module

The product type $A \times B$ is *derived* from the dependent production module `Itt_dprod`. The non-dependent product $A \times B$ is equivalent to $x{:}A \times B$, where x is not free in B.

14.16.1 Parents

include Itt_equal
include Itt_dprod
include Itt_struct

14.16.2 Rewrites

The `unfold_dprod` rewrite unfold the non-dependent product to a dependent-product with a *new* variable x.

! **rewrite** `unfold_prod` : $(A \times B) \longleftrightarrow (x{:}A \times B)$

14.16.3 Rules

Typehood and equality

The product space $A \times B$ is well-formed if both A and B are types.

[2,17]* **rule `independentProductEquality`
{| intro_resource []**;** eqcd_resource **|}** H:
[wf**]** [·] $H \vdash A1 = A2 \in \mathbb{U}$[i] $\longrightarrow$
[wf**]** [·] $H \vdash B1 = B2 \in \mathbb{U}$[i] $\longrightarrow$
[*ext*] $H \vdash (A1 \times B1) = (A2 \times B2) \in \mathbb{U}$[i]
[2,17]* **rule `independentProductType` **{|** intro_resource [] **|}** H:
[wf**]** [·] $H \vdash A1$ Type $\longrightarrow$
[wf**]** [·] $H \vdash A2$ Type $\longrightarrow$
[*ext*] $H \vdash (A1 \times A2)$ Type
[5,29]* **rule `independentProductFormation` H:
A: [*ext*] $H \vdash \mathbb{U}$[i] $\longrightarrow$
B: [*ext*] $H \vdash \mathbb{U}$[i] $\longrightarrow$
[*ext*] $H \vdash \mathbb{U}$[i]

Elimination

The elimination form splits the hypothesis $x: A \times B$ into its parts $u: A$ and $v: B$.

[3,19]* **rule `independentProductElimination`
{| elim_resource [(ThinOption thinT)] **|}** H J z u v:
$t[u; v]$**:** [*ext*] H; z: $A \times B$; u: A; v: B; $J[(u,v)] \vdash T[(u,v)] \longrightarrow$
[*ext*] H; z: $A \times B$; $J[z] \vdash T[z]$

Membership

The members of the non-dependent product $A \times B$ are the pairs (a, b), where $a \in A$ and $b \in B$.

[2,23]* **rule `independentPairEquality`
{| intro_resource []**;** eqcd_resource **|}** H:
[wf**]** [$\cdot$] $H \vdash a1 = a2 \in A \longrightarrow$
[wf**]** [$\cdot$] $H \vdash b1 = b2 \in B \longrightarrow$
[*ext*] $H \vdash (a1,b1) = (a2,b2) \in (A \times B)$

Introduction

The propositional interpretation of the non-dependent product space $A \times B$ is the conjunction $A \wedge B$. The proposition is true if both A and B are true.

[4,30]* **rule `independentPairFormation` **{|** intro_resource [] **|}** H:
a**: [**wf**]** [*ext*] $H \vdash A \longrightarrow$
b**: [**wf**]** [*ext*] $H \vdash B \longrightarrow$
[*ext*] $H \vdash A \times B$

Subtyping

The product space is covariant in both parts.

[3,14]* **rule `independentProductSubtype`
{| intro_resource [] **|}** H:
[$\cdot$] $H \vdash A1 \subseteq A2 \longrightarrow$
[$\cdot$] $H \vdash B1 \subseteq B2 \longrightarrow$
[*ext*] $H \vdash (A1 \times B1) \subseteq (A2 \times B2)$

14.17 Itt_logic module

The `Itt_logic` module defines the propositional interpretations of the basic types. This is a *derived* module. All the propositional connectives are coded in terms of the existing types.

This module also defines several tactics.

14.17.1 Parents

include Itt_equal
include Itt_rfun
include Itt_dfun
include Itt_fun
include Itt_dprod
include Itt_prod
include Itt_union
include Itt_void
include Itt_unit
include Itt_struct

14.17.2 Terms

The following terms define the propositional connectives. The `prop` term defines the space of *propositions* (the same as the type universes).

declare `prop` = Itt_logic!prop[i:l]
declare `true` = Itt_logic!true
declare `false` = Itt_logic!false
declare `not` = Itt_logic!not{'a}
declare `iff` = Itt_logic!iff{'a; 'b}
declare `implies` = Itt_logic!implies{'a; 'b}
declare `and` = Itt_logic!and{'a; 'b}
declare `or` = Itt_logic!or{'a; 'b}
declare `cand` = Itt_logic!cand{'a; 'b}
declare `cor` = Itt_logic!cor{'a; 'b}
declare `all` = Itt_logic!all{'A; x. B['x]}
declare `exists` = Itt_logic!exists{'A; x. B['x]}

14.17.3 Rewrites

The propositional interpretations have the following definitions:

$$
\begin{array}{rcl}
\top & \equiv & \mathit{Unit} \\
\bot & \equiv & \mathit{Void} \\
\neg A & \equiv & A \to \mathit{Void} \\
A \wedge B & \equiv & A \times B \\
A \vee B & \equiv & A + B \\
A \Rightarrow B & \equiv & A \to B \\
A \Leftrightarrow B & \equiv & (A \to B) \wedge (B \to A) \\
\forall x{:}A.B[x] & \equiv & x{:}A \to B[x] \\
\exists x{:}A.B[x] & \equiv & x{:}A \times B[x]
\end{array}
$$

The *conditional* forms $A \wedge_c B$ and $A \vee_c B$ encode the propositional truth from left-to-right.

! **rewrite** `unfold_prop` : Itt_logic!prop[i:l] $\longleftrightarrow$ $\mathbb{U}$[i]
! **rewrite** `unfold_true` : True $\longleftrightarrow$ Unit
! **rewrite** `unfold_false` : False $\longleftrightarrow$ Void
! **rewrite** `unfold_not` : $(\neg a) \longleftrightarrow (a \to \text{Void})$
! **rewrite** `unfold_implies` : $(a \to b) \longleftrightarrow (a \to b)$
! **rewrite** `unfold_iff` : $(a \leftrightarrow b) \longleftrightarrow ((a \to b) \wedge (b \to a))$
! **rewrite** `unfold_and` : $(a \wedge b) \longleftrightarrow (a \times b)$
! **rewrite** `unfold_or` : $(a \vee b) \longleftrightarrow (a + b)$
! **rewrite** `unfold_cand` : $(a \wedge \mathrm{c}\ b) \longleftrightarrow (a \wedge b)$
! **rewrite** `unfold_cor` : $(a \vee \mathrm{c}\ b) \longleftrightarrow (a \vee (\neg a \wedge \mathrm{c}\ b))$
! **rewrite** `unfold_all` : $(\forall x{:}A.\ B[x]) \longleftrightarrow (\mathrm{x}{:}A \to B[x])$
! **rewrite** `unfold_exists` : $(\exists x{:}A.\ B[x]) \longleftrightarrow (x{:}A \times B[x])$

14.17.4 Rules

The rules are divided into groups for each of the propositional connectives. Each of the connectives has a well-formedness rule, and introduction and elimination forms (where possible).

True and False

The `true` and `false` terms are both types. The `true` term is always true; there is no elimination form. The `false` term is always false; there is no introduction form.

[1,7]* **rule `true_univ` **{|** intro_resource []**;** eqcd_resource **|}** H:
  [*ext*] $H \vdash$ True $\in \mathbb{U}$[i]
[1,7]* **rule `true_member` **{|** intro_resource [] **|}** H:
  [*ext*] $H \vdash$ True $\in \mathbb{U}$[i]
[1,7]* **rule `true_type` **{|** intro_resource [] **|}** H:

[*ext*] $H\vdash$ True Type
[1,7]* **rule `true_intro` **{|** intro_resource [] **|}** H:
[*ext*] $H\vdash$ True
[1,7]* **rule `false_univ` **{|** intro_resource []**;** eqcd_resource **|}** H:
[*ext*] $H\vdash$ False $\in$ $\mathbb{U}$[i]
[1,7]* **rule `false_member` **{|** intro_resource [] **|}** H:
[*ext*] $H\vdash$ False $\in$ $\mathbb{U}$[i]
[1,7]* **rule `false_type` **{|** intro_resource [] **|}** H:
[*ext*] $H\vdash$ False Type
[1,5]* **rule `false_elim`
{| elim_resource [(ThinOption thinT)] **|}** H J:
[*ext*] H; x: False; $J[x]\vdash$ $C[x]$
[1,15]* **rule `false_squash` H:
[$\cdot$] $H\vdash$ False $\longrightarrow$
[*ext*] $H\vdash$ False

Negation

The negation ($\neg A$) is a well-formed if A is a type. The negation states that the type A is not inhabited: any proof of A is also a proof of *Void*. To prove the negation, assume A and find a contradiction. The elimination form forms a proof of the goal from a proof of A.

[1,13]* **rule `not_univ` **{|** intro_resource []**;** eqcd_resource **|}** H:
[wf**]** [$\cdot$] $H\vdash$ $t1 = t2 \in \mathbb{U}$[i] $\longrightarrow$
[*ext*] $H\vdash$ $(\neg t1) = (\neg t2) \in \mathbb{U}$[i]
[1,13]* **rule `not_type` **{|** intro_resource [] **|}** H:
[wf**]** [$\cdot$] $H\vdash$ t Type $\longrightarrow$
[*ext*] $H\vdash$ $(\neg t)$ Type
[2,22]* **rule `not_intro` **{|** intro_resource [] **|}** H x:
[wf**]** [$\cdot$] $H\vdash$ t Type $\longrightarrow$
[main**]** [*ext*] H; x: $t\vdash$ False $\longrightarrow$
[*ext*] $H\vdash$ $\neg t$
[4,27]* **rule `not_elim` **{|** elim_resource [] **|}** H J:
[assertion**]** [*ext*] H; x: $\neg t$; $J[x]\vdash$ t $\longrightarrow$
[*ext*] H; x: $\neg t$; $J[x]\vdash$ C

Conjunction

The conjunction ($A \wedge B$) is well-formed if both A and B are types. It is true if both A and B are true. The elimination form splits the assumption into it two component proofs.

[1,13]* **rule `and_univ` **{|** intro_resource []**;** eqcd_resource **|}** H:

[wf**]** [·] $H \vdash a1 = b1 \in \mathbb{U}$[i] $\longrightarrow$
[wf**]** [·] $H \vdash a2 = b2 \in \mathbb{U}$[i] $\longrightarrow$
[*ext*] $H \vdash (a1 \wedge a2) = (b1 \wedge b2) \in \mathbb{U}$[i]

[1,13]* **rule `and_type` **{|** intro_resource [] **|}** H:
[wf**]** [·] $H \vdash a1$ Type $\longrightarrow$
[wf**]** [·] $H \vdash a2$ Type $\longrightarrow$
[*ext*] $H \vdash (a1 \wedge a2)$ Type

[1,13]* **rule `and_intro` **{|** intro_resource [] **|}** H:
[wf**]** [*ext*] $H \vdash a1 \longrightarrow$
[wf**]** [*ext*] $H \vdash a2 \longrightarrow$
[*ext*] $H \vdash a1 \wedge a2$

[1,11]* **rule `and_elim` **{|** elim_resource [] **|}** H J y z:
[main**]** [*ext*] H; y: $a1$; z: $a2$; $J[(y,z)] \vdash C[(y,z)] \longrightarrow$
[*ext*] H; x: $a1 \wedge a2$; $J[x] \vdash C[x]$

Disjunction

The disjunction $(A \vee B)$ is well-formed if both A and B are types. The disjunction is true if it is a type and one of A or B is true. The introduction rules use the `SelectOption` to allow application with the `selT` tactical. The `selT 1 (dT 0)` tactic applies the `or_intro_left` rule, and `selT 2 (dT 0)` applies the `or_intro_right` rule. The elimination rule performs a case analysis on the disjunctive assumption, producing a case for the left proof of A, and another for the right proof of B.

[1,13]* **rule `or_univ` **{|** intro_resource []; eqcd_resource **|}** H:
[wf**]** [·] $H \vdash a1 = b1 \in \mathbb{U}$[i] $\longrightarrow$
[wf**]** [·] $H \vdash a2 = b2 \in \mathbb{U}$[i] $\longrightarrow$
[*ext*] $H \vdash (a1 \vee a2) = (b1 \vee b2) \in \mathbb{U}$[i]

[1,13]* **rule `or_type` **{|** intro_resource [] **|}** H:
[wf**]** [·] $H \vdash a1$ Type $\longrightarrow$
[wf**]** [·] $H \vdash a2$ Type $\longrightarrow$
[*ext*] $H \vdash (a1 \vee a2)$ Type

[1,15]* **rule `or_intro_left`
{| intro_resource [(SelectOption 1)] **|}** H:
[wf**]** [·] $H \vdash a2$ Type $\longrightarrow$
[main**]** [*ext*] $H \vdash a1 \longrightarrow$
[*ext*] $H \vdash a1 \vee a2$

[1,19]* **rule `or_intro_right`
{| intro_resource [(SelectOption 2)] **|}** H:
[wf**]** [·] $H \vdash a1$ Type $\longrightarrow$
[main**]** [*ext*] $H \vdash a2 \longrightarrow$
[*ext*] $H \vdash a1 \vee a2$

[1,15]* **rule `or_elim` **{|** elim_resource [] **|}** H J y:

[main**]** [*ext*] *H*; y: *a1*; *J*[inl *y*]⊢ *C*[inl *y*] ⟶
[main**]** [*ext*] *H*; y: *a2*; *J*[inr *y*]⊢ *C*[inr *y*] ⟶
[*ext*] *H*; x: *a1* ∨ *a2*; *J*[*x*]⊢ *C*[*x*]

Implication

The implication (*A* → *B*) is well-formed if both *A* and *B* are types. The implication is true if it is a type, and a proof of *B* can be produced from a proof of *A*. The elimination rule corresponds to *modus-ponens*: if a proof of *A* can be found, so can a proof of *B*.

[1,13]* **rule `implies_univ`
{| intro_resource []**;** eqcd_resource **|}** H:
[wf**]** [·] *H*⊢ *a1* = *b1* ∈ 𝕌[i] ⟶
[wf**]** [·] *H*⊢ *a2* = *b2* ∈ 𝕌[i] ⟶
[*ext*] *H*⊢ (*a1* → *a2*) = (*b1* → *b2*) ∈ 𝕌[i]
[2,17]* **rule `implies_type` **{|** intro_resource [] **|}** H:
[wf**]** [·] *H*⊢ *a1* Type ⟶
[wf**]** [·] *H*⊢ *a2* Type ⟶
[*ext*] *H*⊢ (*a1* → *a2*) Type
[1,13]* **rule `implies_intro` **{|** intro_resource [] **|}** H x:
[wf**]** [·] *H*⊢ *a1* Type ⟶
[main**]** [*ext*] *H*; x: *a1*⊢ *a2* ⟶
[*ext*] *H*⊢ *a1* → *a2*
[1,23]* **rule `implies_elim`
{| elim_resource [(ThinOption thinT)] **|}** H J y:
[assertion**]** [*ext*] *H*; x: *a1* → *a2*; *J*[*x*]⊢ *a1* ⟶
[main**]** [*ext*] *H*; x: *a1* → *a2*; *J*[*x*]; y: *a2*⊢ *C*[*x*] ⟶
[*ext*] *H*; x: *a1* → *a2*; *J*[*x*]⊢ *C*[*x*]

Bi-implication

The bi-implication (*A* ↔ *B*) is a well-formed if both *A* and *B* are types. The introduction and elimination rules perform the top-level conjunctive reasoning.

[1,25]* **rule `iff_univ` **{|** intro_resource []**;** eqcd_resource **|}** H:
[wf**]** [·] *H*⊢ *a1* = *b1* ∈ 𝕌[i] ⟶
[wf**]** [·] *H*⊢ *a2* = *b2* ∈ 𝕌[i] ⟶
[*ext*] *H*⊢ (*a1* ↔ *a2*) = (*b1* ↔ *b2*) ∈ 𝕌[i]
[1,31]* **rule `iff_type` **{|** intro_resource [] **|}** H:
[wf**]** [·] *H*⊢ *a1* Type ⟶
[wf**]** [·] *H*⊢ *a2* Type ⟶
[*ext*] *H*⊢ (*a1* ↔ *a2*) Type

*[1,23] **rule** iff_intro {| intro_resource [] |} H:
[wf**]** [*ext*] *H*⊢ *a1* → *a2* ⟶
[wf**]** [*ext*] *H*⊢ *a2* → *a1* ⟶
[*ext*] *H*⊢ *a1* ↔ *a2*
*[2,22] **rule** iff_elim {| elim_resource [] |} H J y z:
[*ext*] *H*; y: *a1* → *a2*; z: *a2* → *a1*; *J*[(*y*,*z*)]⊢ *C*[(*y*,*z*)] ⟶
[*ext*] *H*; x: *a1* ↔ *a2*; *J*[*x*]⊢ *C*[*x*]

Conditional conjunction

The conditional conjunction (*A* ∧c *B*) differs from the conjunction only in the introduction rule. The conjunction is true if *A* is true, and a proof of *B* can be produced from a proof of *A*.

*[1,21] **rule** cand_univ
{| intro_resource []; eqcd_resource |} H x:
[wf**]** [·] *H*⊢ *a1* = *b1* ∈ 𝕌[i] ⟶
[wf**]** [·] *H*; x: *a1*⊢ *a2* = *b2* ∈ 𝕌[i] ⟶
[*ext*] *H*⊢ (*a1* ∧c *a2*) = (*b1* ∧c *b2*) ∈ 𝕌[i]
*[1,21] **rule** cand_type {| intro_resource [] |} H x:
[wf**]** [·] *H*⊢ *a1* Type ⟶
[wf**]** [·] *H*; x: *a1*⊢ *a2* Type ⟶
[*ext*] *H*⊢ (*a1* ∧c *a2*) Type
*[5,49] **rule** cand_intro {| intro_resource [] |} H x:
[main**]** [*ext*] *H*⊢ *a1* ⟶
[main**]** [*ext*] *H*; x: *a1*⊢ *a2* ⟶
[*ext*] *H*⊢ *a1* ∧c *a2*
*[2,8] **rule** cand_elim {| elim_resource [] |} H J y z:
[main**]** [*ext*] *H*; y: *a1*; z: *a2*; *J*[(*y*,*z*)]⊢ *C*[(*y*,*z*)] ⟶
[*ext*] *H*; x: *a1* ∧c *a2*; *J*[*x*]⊢ *C*[*x*]

Conditional disjunction

The conditional disjunction (*A* ∨c *B*) differs from the disjunction in that a proof of *B* is needed only if a proof of *A* can't be found. The conditional disjunction is true if *A* is true, or *B* is true *assuming* that *A* is false. The elimination rule produces the two cases, one where there is a proof of *A*, and another where there is a proof of *B* and a proof of falsehood for *A*.

*[1,22] **rule** cor_univ
{| intro_resource []; eqcd_resource |} H x:
[wf**]** [·] *H*⊢ *a1* = *b1* ∈ 𝕌[i] ⟶
[wf**]** [·] *H*; x: ¬*a1*⊢ *a2* = *b2* ∈ 𝕌[i] ⟶

[*ext*] $H \vdash (a1 \vee_c a2) = (b1 \vee_c b2) \in \mathbb{U}$[i]

[1,22]* **rule `cor_type` **{|** intro_resource [] **|}** H x:
[wf**]** [·] $H \vdash a1$ Type ⟶
[wf**]** [·] H; x: $\neg a1 \vdash a2$ Type ⟶
[*ext*] $H \vdash (a1 \vee_c a2)$ Type

[5,34]* **rule `cor_intro_left`
{| intro_resource [(SelectOption 1)] **|}** H x:
[wf**]** [·] H; x: $\neg a1 \vdash a2$ Type ⟶
[main**]** [*ext*] $H \vdash a1$ ⟶
[*ext*] $H \vdash a1 \vee_c a2$

[3,25]* **rule `cor_intro_right`
{| intro_resource [(SelectOption 2)] **|}** H x:
[wf**]** [·] $H \vdash a1$ Type ⟶
[main**]** [*ext*] $H \vdash \neg a1$ ⟶
[main**]** [*ext*] H; x: $\neg a1 \vdash a2$ ⟶
[*ext*] $H \vdash a1 \vee_c a2$

[2,15]* **rule `cor_elim` **{|** elim_resource [] **|}** H J u v:
[main**]** [*ext*] H; u: $a1$; J[inl u] $\vdash C$[inl u] ⟶
[main**]** [*ext*] H; u: $\neg a1$; v: $a2$; J[inr (u,v)] $\vdash C$[inr (u,v)] ⟶
[*ext*] H; x: $a1 \vee_c a2$; J[x] $\vdash C$[x]

Universal quantification

The universal quantification $(\forall x{:}A.\ B[x])$ is well-formed if A is a type, and $B[x]$ is a type for any $x \in A$. The quantification is true if it is well-formed and a $B[a]$ is true for any element $a \in A$. The elimination form allows *instantiation* of quantification on a particular element $a \in A$, to produce a proof of $B[a]$.

[1,13]* **rule `all_univ`
{| intro_resource []**;** eqcd_resource **|}** H x:
[wf**]** [·] $H \vdash t1 = t2 \in \mathbb{U}$[i] ⟶
[wf**]** [·] H; x: $t1 \vdash b1[x] = b2[x] \in \mathbb{U}$[i] ⟶
[*ext*] $H \vdash (\forall x1{:}t1.\ b1[x1]) = (\forall x2{:}t2.\ b2[x2]) \in \mathbb{U}$[i]

[1,13]* **rule `all_type` **{|** intro_resource [] **|}** H x:
[wf**]** [·] $H \vdash t$ Type ⟶
[wf**]** [·] H; x: $t \vdash b[x]$ Type ⟶
[*ext*] $H \vdash (\forall v{:}t.\ b[v])$ Type

[1,13]* **rule `all_intro` **{|** intro_resource [] **|}** H x:
[wf**]** [·] $H \vdash t$ Type ⟶
[main**]** [*ext*] H; x: $t \vdash b[x]$ ⟶
[*ext*] $H \vdash \forall v{:}t.\ b[v]$

[3,32]* **rule `all_elim`
{| elim_resource [(ThinOption thinT)] **|}** H J w z:

[wf**]** [·] H; x: $\forall a{:}A.\ B[a]$; $J[x] \vdash z \in A \longrightarrow$
[main**]** [*ext*] H; x: $\forall a{:}A.\ B[a]$; $J[x]$; w: $B[z] \vdash C[x] \longrightarrow$
[*ext*] H; x: $\forall a{:}A.\ B[a]$; $J[x] \vdash C[x]$

Existential quantification

The existential quantification $(\exists x{:}A.\ B[x])$ is well-formed if A is a type, and $B[x]$ is a type for any $x \in A$. The quantification is true if it is well-formed and there is a proof $a \in A$ where $B[a]$ is also true. The elimination form splits the proof of $\exists x{:}A.B[x]$ into its parts.

[1,13]* **rule `exists_univ`
{| intro_resource []**;** eqcd_resource **|}** H x:
[wf**]** [·] $H \vdash t1 = t2 \in \mathbb{U}$[i] $\longrightarrow$
[wf**]** [·] H; x: $t1 \vdash b1[x] = b2[x] \in \mathbb{U}$[i] $\longrightarrow$
[*ext*] $H \vdash (\exists x1{:}t1.\ b1[x1]) = (\exists x2{:}t2.\ b2[x2]) \in \mathbb{U}$[i]
[1,13]* **rule `exists_type` **{|** intro_resource [] **|}** H x:
[wf**]** [·] $H \vdash t$ Type $\longrightarrow$
[wf**]** [·] H; x: $t \vdash b[x]$ Type $\longrightarrow$
[*ext*] $H \vdash (\exists v{:}t.\ b[v])$ Type
[4,20]* **rule `exists_intro` **{|** intro_resource [] **|}** H z x:
[wf**]** [·] $H \vdash z \in t \longrightarrow$
[main**]** [*ext*] $H \vdash b[z] \longrightarrow$
[wf**]** [·] H; x: $t \vdash b[x]$ Type $\longrightarrow$
[*ext*] $H \vdash \exists v{:}t.\ b[v]$
[1,11]* **rule `exists_elim` **{|** elim_resource [] **|}** H J y z:
[main**]** [*ext*] H; y: a; z: $b[y]$; $J[(y,z)] \vdash C[(y,z)] \longrightarrow$
[*ext*] H; x: $\exists v{:}a.\ b[v]$; $J[x] \vdash C[x]$

14.17.5 Tactics

The `Itt_logic` module defines several tactics for reasoning the the Nuprl type theory. The tactics perform *generic* reasoning of Nuprl sequents.

`moveToConclT`, `moveToConclVarsT` The `moveToConclT` tactic "moves" a hypothesis to the conclusion using the implication form. The generic usage is as follows:

$$\begin{array}{l} [ext]\ H; J \vdash \forall x{:}T_1.T_2 \\ \text{—BY moveToConclT}\,i \\ [ext]\ H; i{:}x{:}T_1; J \vdash T_2. \end{array}$$

The argument i is the index of the hypothesis. In some cases, there may be additional hypotheses following $x\colon T_1$ that *depend* on the hypothesis x. These hypotheses are also moved to the conclusion.

$$\begin{array}{l} [ext]\ H; j\colon \mathbb{Z} \vdash \forall i\colon \mathbb{Z}.(i < j) \Rightarrow T_2[i] \\ \text{—BY moveToConclT}i \\ [ext]\ H; i\colon \mathbb{Z}; j\colon \mathbb{Z}; w\colon i < j \vdash T_2[i] \end{array}$$

The moveToConclVarsT takes a list of hypotheses, identified by their binding variables, to move to the conclusion.

univCDT, genUnivCDT The univCDT and genUnivCDT tactics apply introduction reasoning on the goal. The univCDT tactic decomposes universal quantifications, implications, and function spaces. The genUnivCDT tactic also chains through conjunctions and biconditionals.

$$\begin{array}{l} [ext]\ H; x_1\colon T_1; \cdots; x_n\colon T_n \vdash T_{n+1} \\ [squash]\ H; x_1\colon T_1; \cdots; x_{n-1}\colon T_{n-1} \vdash T_n\ \mathit{Type} \\ \vdots \\ [squash]\ H \vdash T_1\ \mathit{Type} \\ \text{—BY univCDT} \\ [ext]\ H \vdash \forall x_1\colon T_1. \ldots \forall x_n\colon T_n. T_{n+1} \end{array}$$

instHypT The instHypT tactic performs or instantiation of a hypothesis. The hypothesis must be a universal quantification or an implication.

backThruHypT The backThruHypT performs backward-chaining through a hypothesis. The conclusion must match a suffix of the hypothesis, which must be a sequence of universal quantifications or implications through that suffix.

$$\begin{array}{l} [squash]\ H; y\colon \forall x_1\colon T_1. \ldots . T_{n+1}[x_1, \ldots, x_n]; J[y] \vdash t_1 \in T_1 \\ \vdots \\ [squash]\ H; y\colon \forall x_1\colon T_1. \ldots . T_{n+1}[x_1, \ldots, x_n]; J[y] \vdash t_n \in T_n \\ \text{—BY backThruHypT}i \\ []\ H; y\colon \forall x_1\colon T_1. \ldots . T_{n+1}[x_1, \ldots, x_n]; J[y] \vdash T_{n+1}[t_1, \ldots, t_n] \end{array}$$

The backThruHypT computes the argument terms $t_1, \ldots, t_n$ by matching the goal with the hypothesis.

`assumT` *Assumptions* correspond to the subgoals of the outermost theorem statement. The `assumT` tactic instantiates an assumption as a universally quantified hypothesis.

$$
\begin{array}{l}
\quad [\,]\ H;\ldots \vdash T_1 \\
\quad \vdots \\
\quad [\,]\ H; x_1{:}A_1; \cdots; x_n{:}A_n \vdash T_i \\
\quad \vdots \\
\quad [\,]\ H;\ldots \vdash T_m \\
\hline
\quad [\,]\ H; J; w{:}\forall x_1{:}A_1 \ldots . \forall x_n{:}A_n.T_i \vdash C \\
\quad [\,]\ H; J \vdash A_1\ \mathit{Type} \\
\quad \vdots \\
\quad [\,]\ H; J \vdash A_n\ \mathit{Type} \\
\text{—}\texttt{BY assumT}i \\
\quad [\,]\ H;\ldots \vdash T_1 \\
\quad \vdots \\
\quad [\,]\ H; x_1{:}A_1; \cdots; x_n{:}A_n \vdash T_i \\
\quad \vdots \\
\quad [\,]\ H;\ldots \vdash T_m \\
\hline
\quad [\,]\ H; J \vdash C
\end{array}
$$

`backThruAssumT` The `backThruAssumT` performs backward chaining similar to the `backThruHypT`, but on an *assumption*.

`genAssumT` The `genAssumT` generalizes on an assumption.

$$
\begin{array}{l}
\quad [\mathit{ext}]\ H;\ldots \vdash T_1 \\
\quad \vdots \\
\quad [\mathit{ext}]\ H \vdash t \in T_i \\
\quad \vdots \\
\quad [\mathit{ext}]\ H;\ldots \vdash T_n \\
\hline
\quad [\mathit{ext}]\ H \vdash \forall x{:}T_i.C \\
\text{—}\texttt{BY genAssumT}i \\
\quad [\mathit{ext}]\ H;\ldots \vdash T_1 \\
\quad \vdots \\
\quad [\mathit{ext}]\ H \vdash t \in T_i \\
\quad \vdots \\
\quad [\mathit{ext}]\ H;\ldots \vdash T_n \\
\hline
\quad [\mathit{ext}]\ H \vdash C
\end{array}
$$

14.18 Itt_decidable module

It is occasionally useful to assert the *decidability* of a proposition P. The proposition is decidable if $P \vee \neg P$. Decidable propositions allow reasoning by case analysis.

The `Itt_decidable` module defines a generic resource, and a tactic `decideT` to help prove the decidability of propositions.

14.18.1 Parents

include Itt_logic

14.18.2 Terms

declare `decidable` = Itt_decidable!decidable{'p}

14.18.3 Rewrites

The definition of *decidable*(P) if $P \vee \neg P$.

! **rewrite** `unfold_decidable` : Decidable(p) $\longleftrightarrow$ ($p \vee (\neg p)$)

14.18.4 Rules

The `assert_decidable` rule allows case analysis over a decidable proposition p.

[3,16]* **rule `assert_decidable` H p:
[decidable**]** [*ext*] $H \vdash$ Decidable(p) $\longrightarrow$
[*ext*] H; x: $p \vdash G \longrightarrow$
[*ext*] H; x: $\neg p \vdash G \longrightarrow$
[*ext*] $H \vdash G$

14.18.5 Resources

The `decide_resource` provides a resource for reasoning about decidability of propositions. The argument `term * tactic` provides a term that matches the goal to be proved, and a tactic to prove goals of that form.

resource decide_resource**(**(term * tactic)**;**
tactic**;**
decide_data**;**
(Tactic**.**pre_tactic)**)**

14.18.6 Tactics

The decideT tactic applies the assert_decidable on a specific proposition P.

$$\begin{array}{l} [\mathit{squash}]\ H \vdash \mathit{decidable}(P) \\ [\mathit{ext}]\ H; x{:}P \vdash C \\ [\mathit{ext}]\ H; x{:}\neg P \vdash C \\ \text{—BY decideT}P \\ [\mathit{ext}]\ H \vdash C \end{array}$$

Basic decidability

The propositions $\top$ and $\bot$ are always decidable.

*[1,27] **rule** dec_false {| decide_resource |} H:
[*ext*] *H*⊢ Decidable(False)
*[1,22] **rule** dec_true {| decide_resource |} H:
[*ext*] *H*⊢ Decidable(True)

14.19 Itt_bool module

The Itt_bool module defines a type of (decidable) Booleans. The definition of the Boolean values is based on the unit type in the Itt_unit module and the union type in the Itt_union module, as the type *Unit* + *Unit*.

14.19.1 Parents

include Itt_equal
include Itt_struct
include Itt_union
include Itt_set
include Itt_logic

14.19.2 Terms

The following terms define the Boolean connectives. The Boolean values are *not* propositions; the `assert` term converts a Boolean expression to propositional form. Note that these connectives are completely separate from the logical connectives defined in `Itt_logic`.

declare `bool` = Itt_bool!bool
declare `btrue` = Itt_bool!btrue
declare `bfalse` = Itt_bool!bfalse
declare `bor` = Itt_bool!bor{'a; 'b}
declare `band` = Itt_bool!band{'a; 'b}
declare `bimplies` = Itt_bool!bimplies{'a; 'b}
declare `bnot` = Itt_bool!bnot{'a; 'b}
declare `assert` = Itt_bool!assert{'t}
declare `ifthenelse` = Itt_bool!ifthenelse{'e1; 'e2; 'e3}

14.19.3 Rewrites

The *Bool* type is defined as the type *Unit* + *Unit*. The $\top$ term is chosen to be the *left* term, and $\bot$ is the *right* term.

! **rewrite** `unfold_bool` : Bool $\longleftrightarrow$ (Unit + Unit)
! **rewrite** `unfold_btrue` : true $\longleftrightarrow$ (inl $\cdot$)
! **rewrite** `unfold_bfalse` : false $\longleftrightarrow$ (inr $\cdot$)

The `ifthenelse` term is the program that performs case analysis on a Boolean value. The `ifthenelse` term is defined directly in terms of the `decide` term in `Itt_union`.

! **rewrite** `unfold_ifthenelse` :
(if *b* then *e1* else *e2*) $\longleftrightarrow$
match *b* with inl *x* -> *e1* | inr *y* -> *e2*

The Boolean connectives are defined in terms of `ifthenelse`.

! **rewrite** `unfold_bor` : $(a \vee_b b) \longleftrightarrow$ (if *a* then true else *b*)
! **rewrite** `unfold_band` : $(a \wedge_b b) \longleftrightarrow$ (if *a* then *b* else false)
! **rewrite** `unfold_bimplies` : $(a \rightarrow_b b) \longleftrightarrow$ (if *a* then *b* else true)
! **rewrite** `unfold_bnot` : $(\neg_b a) \longleftrightarrow$ (if *a* then false else true)

The *propositional* form of a Boolean value is expressed as an equality judgment.

! **rewrite** `unfold_assert` : $(\downarrow t) \longleftrightarrow (t = \text{true} \in \text{Bool})$

The reductions on literal Booleans are derived from the computational properties of the `union` type.

[1,22]* **rewrite `reduce_ifthenelse_true` :
(if true then *e1* else *e2*) ⟷
e1
[1,22]* **rewrite `reduce_ifthenelse_false` :
(if false then *e1* else *e2*) ⟷
e2

[1,22]* **rewrite `reduce_bnot_true` : (¬*b*true) ⟷ false
[1,22]* **rewrite `reduce_bnot_false` : (¬*b*false) ⟷ true
[1,22]* **rewrite `reduce_bor_true` : (true ∨*b* *e1*) ⟷ true
[1,22]* **rewrite `reduce_bor_false` : (false ∨*b* *e1*) ⟷ *e1*
[1,22]* **rewrite `reduce_band_true` : (true ∧*b* *e1*) ⟷ *e1*
[1,22]* **rewrite `reduce_band_false` : (false ∧*b* *e1*) ⟷ false
[1,22]* **rewrite `reduce_bimplies_true` : (true →*b* *e1*) ⟷ *e1*
[1,22]* **rewrite `reduce_bimplies_false` : (false →*b* *e1*) ⟷ true

14.19.4 Rules

Typehood, well-formedness, and membership

The *Bool* type is a member of every universe, and it contains the terms ⊤ and ⊥.

[1,13]* **rule `boolEquality`
{| intro_resource []**;** eqcd_resource **|}** H:
[*ext*] *H*⊢ Bool ∈ 𝕌[i]
[1,13]* **rule `boolType` **{|** intro_resource [] **|}** H:
[*ext*] *H*⊢ Bool Type
[1,15]* **rule `btrue_member`
{| intro_resource []**;** eqcd_resource **|}** H:
[*ext*] *H*⊢ true ∈ Bool
[1,15]* **rule `bfalse_member`
{| intro_resource []**;** eqcd_resource **|}** H:
[*ext*] *H*⊢ false ∈ Bool

Elimination

The elimination rule performs a case analysis on a Boolean assumption. There are two cases: one where the assumption is true, and the another where it is false.

[4,63]* **rule `boolElimination2` **{|** elim_resource [] **|}** H J x:
[main**]** [*ext*] *H*; *J*[true]⊢ *C*[true] ⟶
[main**]** [*ext*] *H*; *J*[false]⊢ *C*[false] ⟶
[*ext*] *H*; x: Bool; *J*[*x*]⊢ *C*[*x*]

Combinator well-formedness

The `ifthenelse` term computes a type if its argument is Boolean, and its branches are types under a case analysis on the condition.

[10,160]* **rule `ifthenelse_type2` **{|** intro_resource [] **|}** H x:
[wf**]** [·] *H*⊢ *e* ∈ Bool ⟶
[wf**]** [·] *H*; x: *e* = true ∈ Bool⊢ *A* Type ⟶
[wf**]** [·] *H*; x: *e* = false ∈ Bool⊢ *B* Type ⟶
[*ext*] *H*⊢ (if *e* then *A* else *B*) Type

Contradiction

The two following rules represent proof by contradiction: ⊤ and ⊥ are provably distinct.

[5,44]* **rule `boolContradiction1`
{| elim_resource [(ThinOption thinT)] **|}** H J:
[*ext*] *H*; x: true = false ∈ Bool; *J*[*x*]⊢ *C*[*x*]
[5,41]* **rule `boolContradiction2`
{| elim_resource [(ThinOption thinT)] **|}** H J:
[*ext*] *H*; x: false = true ∈ Bool; *J*[*x*]⊢ *C*[*x*]

Combinator equality

The `ifthenelse` term computes a value of type *T* if the condition is a Boolean value, and the branches both have type *T* under a case analysis on the condition.

[23,253]* **rule `ifthenelse_equality`
{| intro_resource []**;** eqcd_resource **|}** H w:
[wf**]** [·] *H*⊢ *e1* = *e2* ∈ Bool ⟶
[wf**]** [·] *H*; w: *e1* = true ∈ Bool⊢ *x1* = *x2* ∈ *T* ⟶
[wf**]** [·] *H*; w: *e1* = false ∈ Bool⊢ *y1* = *y2* ∈ *T* ⟶
[*ext*] *H*⊢ (if *e1* then *x1* else *y1*) = (if *e2* then *x2* else *y2*) ∈ *T*

Computational equivalence

The Boolean values are computationally equivalent if they are equal. This is because the *only* (canonical) terms in *Unit* + *Unit* are the terms $inl(\cdot)$ ($\top$) and $inr(\cdot)$ ($\bot$).

[9,125]* **rule `boolSqequal` H:
[$\cdot$] $H \vdash x = y \in \text{Bool} \longrightarrow$
[*ext*] $H \vdash x \longleftrightarrow y$

Connective well-formedness

The connectives are Boolean values if their immediate subterms are also Boolean values.

[1,24]* **rule `bor_member` **{|** intro_resource [] **|}** H:
[wf**]** [$\cdot$] $H \vdash t1 \in \text{Bool} \longrightarrow$
[wf**]** [$\cdot$] $H \vdash t2 \in \text{Bool} \longrightarrow$
[*ext*] $H \vdash (t1 \vee b\ t2) \in \text{Bool}$
[1,24]* **rule `band_member` **{|** intro_resource [] **|}** H:
[wf**]** [$\cdot$] $H \vdash t1 \in \text{Bool} \longrightarrow$
[wf**]** [$\cdot$] $H \vdash t2 \in \text{Bool} \longrightarrow$
[*ext*] $H \vdash (t1 \wedge b\ t2) \in \text{Bool}$
[1,24]* **rule `bimplies_member` **{|** intro_resource [] **|}** H:
[wf**]** [$\cdot$] $H \vdash t1 \in \text{Bool} \longrightarrow$
[wf**]** [$\cdot$] $H \vdash t2 \in \text{Bool} \longrightarrow$
[*ext*] $H \vdash (t1 \rightarrow b\ t2) \in \text{Bool}$
[1,16]* **rule `bnot_member` **{|** intro_resource [] **|}** H:
[wf**]** [$\cdot$] $H \vdash a \in \text{Bool} \longrightarrow$
[*ext*] $H \vdash (\neg b a) \in \text{Bool}$

Propositional reasoning

The *reasoning* about Boolean expressions is performed using the *propositional* form, coded using the `assert` form. The `assert` term is well-formed if its argument is a Boolean value.

The $\uparrow \top$ goal is always provable; the $\uparrow \bot$ assumption is contradictory.

[1,13]* **rule `assert_type` **{|** intro_resource [] **|}** H:
[wf**]** [$\cdot$] $H \vdash t \in \text{Bool} \longrightarrow$
[*ext*] $H \vdash (\downarrow t)$ Type
[1,7]* **rule `assert_true` **{|** intro_resource [] **|}** H:
[*ext*] $H \vdash \downarrow\text{true}$
[1,6]* **rule `assert_false`

{| elim_resource [(ThinOption thinT)] **|}** H J:
[*ext*] *H*; x: ↓false; *J*[*x*]⊢ *C*[*x*]

Case analysis and substitution

The following two rules perform a case analysis on a Boolean expression in a clause. This reasoning is allowed because the canonical Boolean terms are just ⊤ and ⊥.

[8,237]* **rule `bool_subst_concl` H bind(*x*.*C*[*x*]) *e* y:
[wf**]** [·] *H*⊢ *e* ∈ Bool ⟶
[main**]** [*ext*] *H*; y: ↓*e*⊢ *C*[true] ⟶
[main**]** [*ext*] *H*; y: ↓(¬*be*)⊢ *C*[false] ⟶
[*ext*] *H*⊢ *C*[*e*]
[19,227]* **rule `bool_subst_hyp` H J bind(*x*.*A*[*x*]) *e* y:
[wf**]** [·] *H*; x: *A*[*e*]; *J*[*x*]⊢ *e* ∈ Bool ⟶
[main**]** [*ext*] *H*; x: *A*[true]; *J*[*x*]; y: ↓*e*⊢ *C*[*x*] ⟶
[main**]** [*ext*] *H*; x: *A*[false]; *J*[*x*]; y: ↓(¬*be*)⊢ *C*[*x*] ⟶
[*ext*] *H*; x: *A*[*e*]; *J*[*x*]⊢ *C*[*x*]

Extensional membership

Two Boolean expressions *A* and *B* are equal if the bi-implication $A \Leftrightarrow_b B$ holds.

[8,345]* **rule `bool_ext_equality` H u:
[wf**]** [·] *H*⊢ *x* ∈ Bool ⟶
[wf**]** [·] *H*⊢ *y* ∈ Bool ⟶
[main**]** [·] *H*; u: ↓*x*⊢ ↓*y* ⟶
[main**]** [·] *H*; u: ↓*y*⊢ ↓*x* ⟶
[*ext*] *H*⊢ *x* = *y* ∈ Bool

Reasoning about the Boolean connectives

The following two rules define introduction and elimination reasoning on the Boolean negation.

[9,121]* **rule `assert_bnot_intro` **{|** intro_resource [] **|}** H x:
[wf**]** [·] *H*⊢ *t1* ∈ Bool ⟶
[main**]** [·] *H*; x: ↓*t1*⊢ False ⟶
[*ext*] *H*⊢ ↓(¬*bt1*)
[14,216]* **rule `assert_bnot_elim` **{|** elim_resource [] **|}** H J:
[wf**]** [·] *H*; *J*[·]⊢ ↓*t* ⟶
[*ext*] *H*; x: ↓(¬*bt*); *J*[*x*]⊢ *C*[*x*]

The `magic` rule defines classical reasoning about Boolean values. The rule can be used to prove $\neg\neg A \Rightarrow_b A$, and it can be used to perform *all* Boolean reasoning using only the elimination rules.

[8,158]* **rule `assert_magic` H x:
[wf**]** [·] $H \vdash t \in \text{Bool} \longrightarrow$
[wf**]** [·] H; x: $\downarrow(\neg_b t) \vdash \text{False} \longrightarrow$
[*ext*] $H \vdash \downarrow t$

The following four rules define elimination reasoning on the Boolean binary connectives.

[6,72]* **rule `assert_bor_elim` **{|** elim_resource [] **|}** H J:
[wf**]** [·] H; x: $\downarrow(t1 \vee_b t2)$; $J[x] \vdash t1 \in \text{Bool} \longrightarrow$
[main**]** [*ext*] H; x: $\downarrow t1$; $J[\cdot] \vdash C[\cdot] \longrightarrow$
[main**]** [*ext*] H; x: $\downarrow t2$; $J[\cdot] \vdash C[\cdot] \longrightarrow$
[*ext*] H; x: $\downarrow(t1 \vee_b t2)$; $J[x] \vdash C[x]$
[7,62]* **rule `assert_band_elim` **{|** elim_resource [] **|}** H J y z:
[wf**]** [·] H; x: $\downarrow(t1 \wedge_b t2)$; $J[x] \vdash t1 \in \text{Bool} \longrightarrow$
[main**]** [*ext*] H; y: $\downarrow t1$; z: $\downarrow t2$; $J[\cdot] \vdash C[\cdot] \longrightarrow$
[*ext*] H; x: $\downarrow(t1 \wedge_b t2)$; $J[x] \vdash C[x]$
[9,91]* **rule `assert_bimplies_elim` **{|** elim_resource [] **|}** H J:
[assertion**]** [·] H; $J[\cdot] \vdash \downarrow t1 \longrightarrow$
[main**]** [*ext*] H; $J[\cdot]$; y: $\downarrow t2 \vdash C[\cdot] \longrightarrow$
[*ext*] H; x: $\downarrow(t1 \rightarrow_b t2)$; $J[x] \vdash C[x]$

Finally, we the following rules define introduction reasoning on the Boolean propositional connectives.

[5,37]* **rule `assert_bor_intro_left`
{| intro_resource [(SelectOption 1)] **|}** H:
[wf**]** [·] $H \vdash t2 \in \text{Bool} \longrightarrow$
[main**]** [·] $H \vdash \downarrow t1 \longrightarrow$
[*ext*] $H \vdash \downarrow(t1 \vee_b t2)$
[5,43]* **rule `assert_bor_intro_right`
{| intro_resource [(SelectOption 2)] **|}** H:
[wf**]** [·] $H \vdash t1 \in \text{Bool} \longrightarrow$
[main**]** [·] $H \vdash \downarrow t2 \longrightarrow$
[*ext*] $H \vdash \downarrow(t1 \vee_b t2)$
[7,52]* **rule `assert_band_intro` **{|** intro_resource [] **|}** H:
[main**]** [·] $H \vdash \downarrow t1 \longrightarrow$
[main**]** [·] $H \vdash \downarrow t2 \longrightarrow$
[*ext*] $H \vdash \downarrow(t1 \wedge_b t2)$
[5,40]* **rule `assert_bimplies_intro` **{|** intro_resource [] **|}** H x:
[wf**]** [·] $H \vdash t1 \in \text{Bool} \longrightarrow$

[main**]** [·] H; x: $\downarrow t1 \vdash \downarrow t2 \longrightarrow$
[*ext*] $H \vdash \downarrow(t1 \rightarrow b\ t2)$

Squash reasoning

The proof extract of a Boolean assertion is always the term · term; the proof itself can be omitted.

[1,15]* **rule `assertSquashElim` H:
[·] $H \vdash \downarrow t \longrightarrow$
[*ext*] $H \vdash \downarrow t$

14.19.5 Tactics

`splitBoolT` The `splitBoolT` tactic performs a case analysis on a Boolean expression in a clause. The tactic `splitBoolT i e` produces three subgoals; it asserts that e is actually a Boolean expression, and it produces two subgoals where the expression e in clause i is replaced with the terms $\top$ and $\bot$.

`splitITE` The `splitITE i` tactic searches for an occurrence of a subterm of the form **if** t_1 **then** t_2 **else** t_3 in clause i, and generates two main subgoals, one for the case where t_1 is true and the conditional is replaced with t_2, and the other where t_1 is false and the conditional is replaced with the term t_3.

14.20 Itt_int module

The integers are formalized as a *primitive* type in the Nuprl type theory. Computation over the integers is performed using primitive arithmetic operations in the MetaPRL prover. This encoding is fairly awkward; it is difficult to prove arithmetic statements by referring to the primitive rules. A better solution would be to encode the integers using the recursive type in the `Itt_srec` module.

14.20.1 Parents

include Itt_equal
include Itt_rfun
include Itt_bool

include Itt_logic
include Itt_struct

14.20.2 Terms

The `int` term is the type of integers with elements

$\ldots, number[-2], number[-1], number[0], number[1], number[2], \ldots$

The `ind` term is the induction combinator for building loops indexed by an integer argument.

declare `int` = Itt_int!int
declare `number` = Itt_int!number[n:n]
declare `ind` = Itt_int!ind{'i; m, z. 'down; 'base; m, z. 'up}

The binary arithmetic operators are defined with the following terms.

declare `add` = Itt_int!add{'a; 'b}
declare `sub` = Itt_int!sub{'a; 'b}
declare `mul` = Itt_int!mul{'a; 'b}
declare `div` = Itt_int!div{'a; 'b}
declare `rem` = Itt_int!rem{'a; 'b}
declare `lt` = Itt_int!lt{'a; 'b}
declare `le` = Itt_int!le{'a; 'b}
declare `ge` = Itt_int!ge{'a; 'b}
declare `gt` = Itt_int!gt{'a; 'b}

14.20.3 Rewrites

The `le`, `gt`, and `ge` binary operators are defined in terms the integer order $<$ and equality.

! **rewrite** `unfold_le` : $(a \leq b) \longleftrightarrow ((a < b) \vee (a = b \in \mathbb{Z}))$
! **rewrite** `unfold_gt` : $(a > b) \longleftrightarrow (b < a)$
! **rewrite** `unfold_ge` : $(a \geq b) \longleftrightarrow ((b < a) \vee (a = b \in \mathbb{Z}))$

The binary arithmetic operators are defined using the the *meta* arithmetic operators that are MetaPRL builtin operations.

! **rewrite** `reduce_add` : (i + j) $\longleftrightarrow$ Base_meta!meta_sum{i; j}
! **rewrite** `reduce_sub` : (i - j) $\longleftrightarrow$ Base_meta!meta_diff{i; j}
! **rewrite** `reduce_mul` : (i * j) $\longleftrightarrow$ Base_meta!meta_prod{i; j}
! **rewrite** `reduce_div` : (i$\div$j) $\longleftrightarrow$ Base_meta!meta_quot{i; j}

! **rewrite** `reduce_rem` : (i % j) ⟷ Base_meta!meta_rem{i; j}
! **rewrite** `reduce_lt` : (i < j) ⟷ Base_meta!meta_lt{i; j}
! **rewrite** `reduce_eq` : (i = j ∈ ℤ) ⟷ Base_meta!meta_eq{i; j}

Reduction of the induction combinator `ind` has three cases. If the argument x is 0, the combinator reduces to the *base* case; if it is positive, it reduces to the *up* case; and if it is negative, it reduces to the *down* case. The first argument in the *up* and *down* cases represents the induction value, and the second argument represents the "next" computational step.

14.20.4 Rules

Typehood and well-formedness

The ℤ type inhabits every universe, and it is a type.

! **rule** `intType` **{|** intro_resource [] **|}** H:
[*ext*] *H*⊢ ℤ Type
= ·
! **rule** `intEquality` **{|** intro_resource []**;** eqcd_resource **|}** H:
[*ext*] *H*⊢ ℤ ∈ 𝕌[i]
= ·

Membership

The ℤ type contains the `number` terms.

! **rule** `numberEquality` **{|** intro_resource []**;** eqcd_resource **|}** H:
[*ext*] *H*⊢ n ∈ ℤ
= ·

Elimination

Induction on an integer assumption produces three cases: one for the base case 0, one for induction on negative arguments, and another for induction on positive arguments. The proof extract term uses the `ind` term, which performs a case analysis on its argument.

! **rule** `intElimination`
{| elim_resource [(ThinOption thinT)] **|}** H J n m v z:
down[*n*; *m*; *v*; *z*]**:** [main] [*ext*]
1. *H*

2. n: $\mathbb{Z}$
3. $J[n]$
4. m: $\mathbb{Z}$
5. v: $m < 0$
6. z: $C[m + 1]$
$\vdash C[m] \longrightarrow$

base[*n*]: [main] [*ext*] H; n: $\mathbb{Z}$; $J[n] \vdash C[0] \longrightarrow$

up[*n*; *m*; *v*; *z*]: [main] [*ext*]
1. H
2. n: $\mathbb{Z}$
3. $J[n]$
4. m: $\mathbb{Z}$
5. v: $0 < m$
6. z: $C[m - 1]$
$\vdash C[m] \longrightarrow$

[*ext*] H; n: $\mathbb{Z}$; $J[n] \vdash C[n]$
= Itt_int!ind
{*n*;
m, z. *down*[*n*; *m*; ·; *z*];
base[*n*];
m, z. *up*[*n*; *m*; ·; *z*]}

! **rule** indEquality
{| intro_resource []; eqcd_resource |} H ($\lambda z.T[z]$) x y w:
[wf] [·] $H \vdash x1 = x2 \in \mathbb{Z} \longrightarrow$
[wf] [·]
1. H
2. x: $\mathbb{Z}$
3. w: $x < 0$
4. y: $T[x$ Itt_int!add 1]
$\vdash down1[x; y] = down2[x; y] \in T[x] \longrightarrow$

[wf] [·] $H \vdash base1 = base2 \in T[0] \longrightarrow$
[wf] [·]
1. H
2. x: $\mathbb{Z}$
3. w: $x > 0$
4. y: $T[x$ Itt_int!sub 1]
$\vdash up1[x; y] = up2[x; y] \in T[x] \longrightarrow$

[*ext*]
1. H
$\vdash$
Itt_int!ind
{*x1*;
i1, j1. *down1*[*i1*; *j1*];

base1;
k1, l1. *up1*[*k1*; *l1*]}
= Itt_int!ind
{*x2*;
i2, j2. *down2*[*i2*; *j2*];
base2;
k2, l2. *up2*[*k2*; *l2*]} ∈ *T*[*x1*]
= ·

Well-formedness of the arithmetic operators

The < binary operator is a type if its arguments are integers, and it is inhabited by the · term if it is true.

! **rule** `less_thanEquality`
{| intro_resource []**;** eqcd_resource **|}** H:
[wf**]** [·] *H*⊢ *i1* = *j1* ∈ ℤ ⟶
[wf**]** [·] *H*⊢ *i2* = *j2* ∈ ℤ ⟶
[*ext*] *H*⊢ (*i1* < *j1*) = (*i2* < *j2*) ∈ 𝕌[i]
= ·
! **rule** `less_than_memberEquality`
{| intro_resource []**;** eqcd_resource **|}** H:
[wf**]** [·] *H*⊢ *a* < *b* ⟶
[*ext*] *H*⊢ · ∈ (*a* < *b*)
= ·

The well-formedness of the other binary operators is derived by induction on the integers.

14.21 Itt_isect module

The `Itt_isect` module defines the *intersection* type $\bigcap_{x:A} B[x]$. The elements of the intersection are the terms that inhabit $B[x]$ for *every* $x \in A$. The intersection is similar to the function space $x{:}A \to B[x]$; the intersection is inhabited if-and-only-if there is a constant function in the dependent-function space.

The intersection does not have a conventional set-theoretic interpretation. One example is the type $Top \equiv \bigcap_{x:Void} Void$. If the set theoretic interpretation of *Void* is the empty set, the intersection would probably be empty. However, in the type theory, the intersection contains *every* term t because the quantification is empty.

Another example is the type $\bigcap_{T:\mathbb{U}_i} T \to T$, which contains all the identity functions. The set-theoretic interpretation of functions as sets of ordered pairs would again be empty.

The intersection is commonly used to express polymorphism of function spaces, and it is also used as an operator for record type concatenation. If records are expressed as functions from labels (*Atom* is commonly used for labels) to values, the record type $\{l\colon T\}$ denotes the functions that return values of type T when applied to the label l. The intersection of two record types $\{l_1\colon T_1\} \bigcap \{l_2\colon T_2\}$ is isomorphic to the record space $\{l_1\colon T_1; l_2\colon T_2\}$.

14.21.1 Parents

include Itt_equal
include Itt_set
include Itt_rfun
include Itt_logic

14.21.2 Terms

The `isect` term denotes the intersection type. The `top` type defines the type of all terms $\bigcap_{x:Void} Void$.

declare `isect` = Itt_isect!isect{'A; x. B['x]}
! **rewrite** `unfold_top` : Top $\longleftrightarrow$ $\cap x$:Void.Void

14.21.3 Rules

Typehood and equality

The intersection $\bigcap_{x:A} B[x]$ is well-formed if A is a type, and $B[x]$ is a family of types indexed by $x \in A$.

! **rule** `intersectionEquality`
{| intro_resource []**;** eqcd_resource **|}** H y:
[wf**]** [·] $H \vdash A1 = A2 \in \mathbb{U}$[i] $\longrightarrow$
[wf**]** [·] H; y: $A1 \vdash B1[y] = B2[y] \in \mathbb{U}$[i] $\longrightarrow$
[*ext*] $H \vdash \cap x1{:}A1.B1[x1] = \cap x2{:}A2.B2[x2] \in \mathbb{U}$[i]
= ·
! **rule** `intersectionType` **{|** intro_resource [] **|}** H y:
[wf**]** [·] $H \vdash A$ Type $\longrightarrow$

[wf**]** [·] H; y: A⊢ $B[y]$ Type ⟶
[*ext*] H⊢ ∩x:A.$B[x]$ Type
= ·

The well-formedness of the *Top* type is derived. The *Top* type is a member of every type universe.

[1,14]* **rule `topUniv` **{|** intro_resource [] **|}** H:
[*ext*] H⊢ Top ∈ 𝕌[i]
[1,14]* **rule `topType` **{|** intro_resource [] **|}** H:
[*ext*] H⊢ Top Type

Membership

The elements in the intersection $\bigcap_{x:A} B[x]$ are the terms b that inhabit $B[z]$ for *every* $a \in A$. The member equality of the intersection is the intersection of the equalities on each type $B[z]$. That is, for two elements of the intersection to be equal, they must be equal in *every* type $B[z]$.

The `top` type contains every program. The equality here is trivial; all terms are equal in *Top*.

! **rule** `intersectionMemberEquality`
{| intro_resource []**;** eqcd_resource **|}** H z:
[wf**]** [·] H⊢ A Type ⟶
[wf**]** [·] H; z: A⊢ $b1 = b2 \in B[z]$ ⟶
[*ext*] H⊢ $b1 = b2$ ∈ ∩x:A.$B[x]$
= ·
[1,14]* **rule `topMemberEquality`
{| intro_resource []**;** eqcd_resource **|}** H:
[*ext*] H⊢ $b1 = b2$ ∈ Top

The case equality works in the opposite direction: if two terms are equal in the intersection, they are also equal in each of the case of the intersection.

! **rule** `intersectionMemberCaseEquality` H ∩x:A.$B[x]$ a:
[wf**]** [·] H⊢ $b1 = b2$ ∈ ∩x:A.$B[x]$ ⟶
[wf**]** [·] H⊢ $a \in A$ ⟶
[*ext*] H⊢ $b1 = b2 \in B[a]$
= ·

Elimination

The elimination form performs instantiation of the intersection space. If $a \in A$, the elimination form instantiates the intersection assumption x:$\bigcap_{u:A} B[y]$ to get a proof that $x \in B[a]$.

! **rule** intersectionElimination
{| elim_resource [] **|}** H J *a* x y v:
[wf**]** [·] *H*; x: ∩*y*:*A*.*B*[*y*]; *J*[*x*]⊢ *a* ∈ *A* ⟶
t[*x*; *y*; *v*]**:** **[**main**]** [*ext*]
1. *H*
2. x: ∩*y*:*A*.*B*[*y*]
3. *J*[*x*]
4. z: *B*[*a*]
5. v: *z* = *x* ∈ *B*[*a*]
⊢ *T*[*x*] ⟶
[*ext*] *H*; x: ∩*y*:*A*.*B*[*y*]; *J*[*x*]⊢ *T*[*x*]
= *t*[*x*; *x*; ·]

Subtyping

The intersection type conforms to the subtyping properties of the dependent-function space. The type is *contravariant* in the quantified type *A*, and pointwise-covariant in the domain type *B*[*a*] for each *a* ∈ *A*.

! **rule** intersectionSubtype **{|** intro_resource [] **|}** H a:
[·] *H*⊢ *A2* ⊆ *A1* ⟶
[·] *H*; a: *A1*⊢ *B1*[*a*] ⊆ *B2*[*a*] ⟶
[*ext*] *H*⊢ ∩*a1*:*A1*.*B1*[*a1*] ⊆ ∩*a2*:*A2*.*B2*[*a2*]
= ·

The alternate subtyping rules are for cases where one of the types is not an intersection. The intersection is a subtype of another type *T* if *one* of the cases *B*[*a*] ⊆ *T*. A type *T* is a subtype of the intersection if it is a subtype of *every* case *B*[*a*] for *a* ∈ *A*.

[9,73]* **rule intersectionSubtype2 H y *a*:
[·] *H*⊢ *a* ∈ *A* ⟶
[·] *H*; y: *A*⊢ *B*[*y*] Type ⟶
[·] *H*⊢ *B*[*a*] ⊆ *T* ⟶
[*ext*] *H*⊢ ∩*x*:*A*.*B*[*x*] ⊆ *T*
[5,81]* **rule intersectionSubtype3 H x:
[·] *H*⊢ *A* Type ⟶
[·] *H*⊢ *C* Type ⟶
[·] *H*; x: *A*⊢ *C* ⊆ *B*[*x*] ⟶
[*ext*] *H*⊢ *C* ⊆ ∩*x*:*A*.*B*[*x*]

Every type is a subtype of *Top*.

[1,12]* **rule topSubtype **{|** intro_resource [] **|}** H:
[·] *H*⊢ *T* Type ⟶
[*ext*] *H*⊢ *T* ⊆ Top

14.22 Itt_bisect module

The `Itt_bisect` module derives a binary intersection $A \cap B$ from the intersection `isect` defined in the `Itt_isect` theory, and the Boolean values defined in the `Itt_bool` theory.

14.22.1 Parents

include Itt_isect
include Itt_bool

14.22.2 Terms

declare `bisect` = Itt_bisect!bisect{'A; 'B}

14.22.3 Rewrites

The definition of the binary intersection $A \cap B$ is an intersection over the Booleans.

! **rewrite** `unfold_bisect` : $(A \cap B) \longleftrightarrow \cap x$:Bool.(if x then A else B)

14.22.4 Rules

Typehood and equality

The binary intersection $A \cap B$ is well-formed if both A and B are types.

[2,44]* **rule `bisectEquality`
{| intro_resource []**;** eqcd_resource **|}** H:
[wf**]** [·] $H \vdash A1 = A2 \in \mathbb{U}$[i] $\longrightarrow$
[wf**]** [·] $H \vdash B1 = B2 \in \mathbb{U}$[i] $\longrightarrow$
[*ext*] $H \vdash (A1 \cap B1) = (A2 \cap B2) \in \mathbb{U}$[i]
[2,44]* **rule `bisectType` **{|** intro_resource [] **|}** H:
[wf**]** [·] $H \vdash A$ Type $\longrightarrow$
[wf**]** [·] $H \vdash B$ Type $\longrightarrow$
[*ext*] $H \vdash (A \cap B)$ Type

Membership

Two terms x and y are equal in the binary intersection $A \cap B$ if they are equal in both A and B. Put another way, the elements of the binary intersection are the terms that are members of both A and B.

[5,24]* **rule `bisectMemberEquality`
{| intro_resource []**;** eqcd_resource **|}** H:
[wf**]** [·] $H \vdash x = y \in A \longrightarrow$
[wf**]** [·] $H \vdash x = y \in B \longrightarrow$
[*ext*] $H \vdash x = y \in (A \cap B)$

Elimination

The elimination rule has two forms for an assumption $x{:}A \cap B$. The first produces a witness that $x \in A$, and the second produces a witness for $x \in B$.

[4,31]* **rule `bisectEliminationLeft` H J y z:
[*ext*] H; x: $A \cap B$; $J[x]$; y: A; z: $y = x \in A \vdash C[x] \longrightarrow$
[*ext*] H; x: $A \cap B$; $J[x] \vdash C[x]$
[4,31]* **rule `bisectEliminationRight` H J y z:
[*ext*] H; x: $A \cap B$; $J[x]$; y: B; z: $y = x \in B \vdash C[x] \longrightarrow$
[*ext*] H; x: $A \cap B$; $J[x] \vdash C[x]$

Subtyping

The binary intersection $A \cap B$ is covariant in both A and B.

[8,68]* **rule `bisectSubtypeLeft` H:
[·] $H \vdash B$ Type $\longrightarrow$
[·] $H \vdash A \subseteq C \longrightarrow$
[*ext*] $H \vdash (A \cap B) \subseteq C$
[6,62]* **rule `bisectSubtypeRight` H:
[·] $H \vdash A$ Type $\longrightarrow$
[·] $H \vdash B \subseteq C \longrightarrow$
[*ext*] $H \vdash (A \cap B) \subseteq C$
[5,77]* **rule `bisectSubtypeBelow` H:
[·] $H \vdash C \subseteq A \longrightarrow$
[·] $H \vdash C \subseteq B \longrightarrow$
[*ext*] $H \vdash C \subseteq (A \cap B)$

14.23 Itt_tunion module

The `Itt_tunion` module defines a (joint) union type $\bigcup_{x:A} B[x]$. The elements of the union are the elements of any of the types $B[x]$ for any $x \in A$.

The membership equality is the *union* of the equalities in each of the cases $B[x]$. That is, two elements are equal in the union if they are equal in *any* of the cases. This may be surprising. For example, the type

$$\bigcup_{T:\mathbb{U}_i} T \to T$$

may seem to contain all of the polymorphic functions with type $T \to T$, for any $T \in \mathbb{U}_i$. It does–but the union also contains the type case $\mathit{Void} \to \mathit{Void}$, in which all functions are equal. As a consequence, the union space also has the trivial equality, and thus it has no useful elimination form.

14.23.1 Parents

include Itt_struct
include Itt_equal
include Itt_set

14.23.2 Terms

The `tunion` term defines the union type.

declare `tunion` = Itt_tunion!tunion{'A; x. B['x]}

14.23.3 Rules

Typehood and equality

The union type $\bigcup_{x:A} B[x]$ is well-formed if A is a type, and $B[a]$ is a type for any $a \in A$.

! **rule** `tunionEquality`
{| intro_resource []**;** eqcd_resource **|}** H x:
[wf**]** [·] $H \vdash A1 = A2 \in \mathbb{U}$[i] ⟶
[wf**]** [·] H; x: $A1 \vdash B1[x] = B2[x] \in \mathbb{U}$[i] ⟶
[*ext*] $H \vdash \cup x1{:}A1.B1[x1] = \cup x2{:}A2.B2[x2] \in \mathbb{U}$[i]

= ·

! **rule** tunionType **{|** intro_resource [] **|}** H y:
[wf**]** [·] $H \vdash A$ Type ⟶
[wf**]** [·] H; y: $A \vdash B[y]$ Type ⟶
[*ext*] $H \vdash \cup x{:}A.B[x]$ Type
= ·

Membership

The elements t of the union type are the elements in any one of the branches $t \in B[a]$ for any $a \in A$. Two elements are equal if they are equal in *any* of the branches $B[a]$.

! **rule** tunionMemberEquality
{| intro_resource []**;** eqcd_resource **|}** H a y:
[wf**]** [·] $H \vdash a \in A$ ⟶
[wf**]** [·] H; y: $A \vdash B[y]$ Type ⟶
[wf**]** [·] $H \vdash x1 = x2 \in B[a]$ ⟶
[*ext*] $H \vdash x1 = x2 \in \cup x{:}A.B[x]$
= ·

Introduction

The propositional interpretation of the union type is similar to the existential $\exists x{:}A.B[x]$. The union is inhabited if-and-only-if the existential is also inhabited.

! **rule** tunionMemberFormation **{|** intro_resource [] **|}** H y a:
[wf**]** [·] $H \vdash a \in A$ ⟶
[wf**]** [·] H; y: $A \vdash B[y]$ Type ⟶
t**:** **[**main**]** [*ext*] $H \vdash B[a]$ ⟶
[*ext*] $H \vdash \cup x{:}A.B[x]$
= t

Elimination

The elimination form is weak. The desired rule would be that if, If $x{:}\bigcup_{y:A} B[y]$, then $x \in B[a]$ for some $a \in A$. This rule is allowed, but only for equality goals, where the computational content of the proof can be omitted.

! **rule** tunionElimination
{| elim_resource [(ThinOption thinT)] **|}** H J x w z:
[·] H; x: $\cup y{:}A.B[y]$; $J[x]$; w: A; z: $B[w] \vdash t1[z] = t2[z] \in C[z]$ ⟶

[*ext*] *H*; x: ∪*y*:*A*.*B*[*y*]; *J*[*x*]⊢ *t1*[*x*] = *t2*[*x*] ∈ *C*[*x*]
= ·

14.24 Itt_bunion module

The `Itt_bunion` module defines a binary union $A \cup B$ of type types A and B. The elements include the elements of A as well as the elements of B. Two elements are equal if they are equal in *either* of the types.

14.24.1 Parents

include Itt_tunion
include Itt_bool
include Itt_struct

14.24.2 Terms

declare `bunion` = Itt_bunion!bunion{'A; 'B}

14.24.3 Rewrites

The binary union is defined using the `tunion` over the space of Booleans.

! **rewrite** `unfold_bunion` : $(A \cup B) \longleftrightarrow (\cup x{:}\text{Bool}.(\text{if } x \text{ then } A \text{ else } B))$

14.24.4 Rules

Typehood and equality

The union $A \cup B$ is well-formed if both A and B are types.

[2,44]* **rule `bunionEquality`
{| intro_resource []**;** eqcd_resource **|}** H:
[wf**]** [·] *H*⊢ *A1* = *A2* ∈ 𝕌[i] ⟶
[wf**]** [·] *H*⊢ *B1* = *B2* ∈ 𝕌[i] ⟶
[*ext*] *H*⊢ (*A1* ∪ *B1*) = (*A2* ∪ *B2*) ∈ 𝕌[i]
[2,44]* **rule `bunionType` **{|** intro_resource [] **|}** H:
[wf**]** [·] *H*⊢ *A* Type ⟶
[wf**]** [·] *H*⊢ *B* Type ⟶
[*ext*] *H*⊢ (*A* ∪ *B*) Type

Membership

Two terms are equal in the binary union if they are equal in either type.

[4,64]* **rule `bunionMemberEqualityLeft`
{| intro_resource [(SelectOption 1)]**;** eqcd_resource **|}** H:
[wf**]** [·] $H \vdash x = y \in A \longrightarrow$
[wf**]** [·] $H \vdash B$ Type $\longrightarrow$
[*ext*] $H \vdash x = y \in (A \cup B)$
[5,60]* **rule `bunionMemberEqualityRight`
{| intro_resource [(SelectOption 2)]**;** eqcd_resource **|}** H:
[wf**]** [·] $H \vdash x = y \in B \longrightarrow$
[wf**]** [·] $H \vdash A$ Type $\longrightarrow$
[*ext*] $H \vdash x = y \in (A \cup B)$

Elimination

The elimination form retains the limitations of the general union elimination `tunionElimination`: it can be used only for equality judgments. The elimination form for a union type $A \cup B$ produces two cases: one for membership in A, and another for membership in B.

[2,33]* **rule `bunionElimination`
{| elim_resource [(ThinOption thinT)] **|}** H J y:
[main**]** [·] H; x: $A \cup B$; $J[x]$; y: $A \vdash t1[y] = t2[y] \in C[y] \longrightarrow$
[main**]** [·] H; x: $A \cup B$; $J[x]$; y: $B \vdash t1[y] = t2[y] \in C[y] \longrightarrow$
[*ext*] H; x: $A \cup B$; $J[x] \vdash t1[x] = t2[x] \in C[x]$

14.25 Itt_w module

The `Itt_w` module defines the recursive W-type. The W type has the form $W(x{:}A.B[x])$, where A is a type, and $B[a]$ is a family of types indexed by $a \in A$.

The elements of the W-type are the nodes $tree(a; f)$, where $a \in A$, and f is a function with domain $B[a]$ that produces the "children" of the node. The children also inhabit the W-type, and f is required to have type $B[a] \to W(x{:}A.B[x])$.

The W-type is defined as primitive. It *can* be derived from the recursive type `Itt_srec`, with the definition:

$$W(x{:}A.B[x]) \equiv \mu(T.x{:}A \times B[a] \to T).$$

However, the W type has a simpler semantics than the recursive type. We keep it as primitive so that the recursive type can be omitted if the semantics are questionable.

14.25.1 Parents

include Itt_equal
include Itt_rfun
include Itt_struct

14.25.2 Terms

The W type is type of trees, $W = a{:}\, A \times B[a] \to W$.

declare `w` = Itt_w!w{'A; x. B['x]}

The `tree` terms inhabit the W-type. Each node has a label $a \in A$, and a function f with domain $B[a]$ to compute the children of the node.

declare `tree` = Itt_w!tree{'a; 'f}

The `tree_ind` term is the induction combinator, which provides computation over trees.

declare `tree_ind` = Itt_w!tree_ind{'z; a, f, g. body['a; 'f; 'g]}

14.25.3 Rewrites

The induction combinator takes are W-node as its first argument, and a body that expect three arguments a_2, f_2, and g_2. The a_2 argument is the label of the current node, the f_2 argument is the function that computes the children, and g_2 is the value that is returned by a recursive call.

! **rewrite** `reduce_tree_ind` :
    (tree_ind(*g2*. let tree(*a2*, *f2*) = tree(*a1*, *f1*) in
        body[*a2*; *f2*; *g2*]) ⟷
    body[*a1*;
        f1;
        (λ*a*.(tree_ind(*g2*. let tree(*a2*, *f2*) = *f1 a* in
            body[*a2*; *f2*; *g2*]))]

Trees admit eta-reduction on the function argument—the guarantee is that the function argument will only be used by application, not by any other intensional analysis.

! **rewrite** `reduce_tree_eta` : tree(a, ($\lambda x.f\ x$)) $\longleftrightarrow$ tree(a, f)

14.25.4 Rules

Typehood and equality

The W-type $W(x{:}A.B[x])$ is well-formed if A is a type, and $B[a]$ is a type for any $a \in A$.

! **rule** `wEquality` **{|** intro_resource []**;** eqcd_resource **|}** H y:
[wf**]** [$\cdot$] $H \vdash A1 = A2 \in \mathbb{U}$[i] $\longrightarrow$
[wf**]** [$\cdot$] H; y: $A1 \vdash B1[y] = B2[y] \in \mathbb{U}$[i] $\longrightarrow$
[*ext*] $H \vdash (\mathbb{W}x1{:}A1.B1[x1]) = (\mathbb{W}x2{:}A2.B2[x2]) \in \mathbb{U}$[i]
= $\cdot$

! **rule** `wType` **{|** intro_resource [] **|}** H x:
[wf**]** [$\cdot$] $H \vdash A1$ Type $\longrightarrow$
[wf**]** [$\cdot$] H; x: $A1 \vdash A2[x]$ Type $\longrightarrow$
[*ext*] $H \vdash (\mathbb{W}y{:}A1.A2[y])$ Type
= $\cdot$

Membership

The elements of the W-type $W(x{:}A.B[x])$ are the `tree` terms $tree(a; f)$, where $a \in A$, and $f \in B[a] \to W(x{:}A.B[x])$.

! **rule** `treeEquality` **{|** intro_resource []**;** eqcd_resource **|}** H y:
[wf**]** [$\cdot$] $H \vdash a1 = a2 \in A$ $\longrightarrow$
[wf**]** [$\cdot$] $H \vdash b1 = b2 \in (B[a1] \to (\mathbb{W}x{:}A.B[x]))$ $\longrightarrow$
[wf**]** [$\cdot$] H; y: $A \vdash B[y]$ Type $\longrightarrow$
[*ext*] $H \vdash$ tree($a1$, $b1$) = tree($a2$, $b2$) $\in (\mathbb{W}x{:}A.B[x])$
= $\cdot$

Elimination

The elimination rule performs induction over the W-type $W(x{:}A.B[x])$. The conclusion is true for all z in the W-type if it is true for an arbitrary term $tree(a; f)$, where the induction hypothesis holds on all children given by f.

! **rule** `wElimination`
{| elim_resource [(ThinOption thinT)] **|}** H J z a f g b v:
$t[z; a; f; g]$**:** **[**main**]** [*ext*]
1. H
2. z: $\mathbb{W}x{:}A.B[x]$
3. $J[z]$
4. a: A
5. f: $B[a] \to (\mathbb{W}x{:}A.B[x])$
6. g: b:$B[a] \to T[(f\ b)]$
7. v: $z = \text{tree}(a, f) \in (\mathbb{W}x{:}A.B[x])$

$\vdash T[\text{tree}(a, f)] \longrightarrow$
[*ext*] H; z: $\mathbb{W}x{:}A.B[x]$; $J[z] \vdash T[z]$
$= (\text{tree_ind}(g.\ \text{let tree}(a, f) = z \text{ in } t[z; a; f; g])$

Combinator equality

The tree-induction term computes a value of type T if the body computes a value of type T given and argument $a \in A$, a child function f, and a function g that computes the recursive values for each of the children.

! **rule** `tree_indEquality`
{| intro_resource **[]; |}** eqcd_resource **|}** H $(\mathbb{W}x{:}A.B[x])$:
[wf**]** $[\cdot]$ $H \vdash z1 = z2 \in (\mathbb{W}x{:}A.B[x]) \longrightarrow$
[wf**]** $[\cdot]$
1. H
2. a1: A
3. f1: $B[a1] \to (\mathbb{W}x{:}A.B[x])$
4. g1: x:$A \to B[x] \to T$

$\vdash body1[a1; f1; g1] = body2[a1; f1; g1] \in T \longrightarrow$
[*ext*]
1. H

$\vdash$
$(\text{tree_ind}(g1.\ \text{let tree}(a1, f1) = z1 \text{ in } body1[a1; f1; g1])$
$= (\text{tree_ind}(g2.\ \text{let tree}(a2, f2) = z2 \text{ in } body2[a2; f2; g2])$
$\in T$
$= \cdot$

14.26 Itt_prec module

The `Itt_prec` module define the *parameterized* recursive type. The parameter allows values to be passed as the recursive type is unrolled. The

syntax of the type is $\mu(T, x.B[T, x]; a)$, there T is the type that is being defined, x represents the parameter, with initial value a, and $B[T, x]$ is the definition of the type. The body $B[T, x]$ must be monotone in the type argument T.

The following type definition defines the list of strictly increasing positive integers.

$$\mu(T, i.Unit + k\colon \{j\colon \mathbb{Z} \mid j > i\} \times T(k); 0).$$

The *nil* element is $inl(\cdot)$, and the *cons* operation is the right injection contains a pair of an integer k that is larger that the parameter and a list increasing integers larger than k.

14.26.1 Parents

include Itt_equal
include Itt_subtype
include Itt_void
include Itt_fun
include Itt_prod
include Itt_struct

14.26.2 Terms

The `prec` term defines the parameterized recursive type. The `precind` term is the induction combinator, for computation over the elements in the recursive type.

declare `prec` = Itt_prec!prec{T, x. B['T; 'x]; 'a}
declare `precind` = Itt_prec!precind{'a; p, h. g['p; 'h]}

14.26.3 Rewrites

The `precind` term takes two arguments. The argument a is the term over which the computation is being performed, and the $g[p, h]$ term defines the body of the computation. The first parameter in the body, p, represents the result of the recursive computation, and the second parameter h represents the argument a itself.

! **rewrite** `reducePrecind` :
precind(a; p,p. g[p; h] $\longleftrightarrow$
g[(λa.precind(a; p,p. g[p; h]); a]

14.26.4 Rules

Equality

The `prec` type $\mu(T, x.B[T, x]; a)$ is well-formed if: 1) there is a type of parameters A; 2) the initial parameter a has type A; and 3) the body $B[T, x]$ is well-formed for any argument $x \in A$ and *any* type $T \in \mathbb{U}_i$. In addition, $B[T, x]$ must be *monotone* in the type argument T. If $T_1 \subseteq T_2$, then $B[T_1, x] \subseteq B[T_2, x]$.

! **rule** `precEquality`
{| intro_resource []**;** eqcd_resource **|}** H *A* x y z T P1 P2:
[wf**]** [·] $H \vdash a1 = a2 \in A \longrightarrow$
[wf**]** [·] H; x: A; T: $A \to \mathbb{U}$[i]$\vdash B1[T; x] = B2[T; x] \in \mathbb{U}$[i] $\longrightarrow$
[wf**]** [·]
1. H
2. P1: $A \to \mathbb{U}$[i]
3. P2: $A \to \mathbb{U}$[i]
4. z: x:$A \to ((P1\ x) \subseteq (P2\ x))$
5. x: A
6. y: $B1[P1; x]$
$\vdash y \in B1[P2; x] \longrightarrow$

$H \vdash \mu\{A1,x1.B1[A1; x1]; a1\} = \mu\{A2,x2.B2[A2; x2]; a2\} \in \mathbb{U}$[i]
= ·

Membership

The elements of the parameterized recursive type $\mu(T, x.B[T, x]; a)$ are the elements in the body $B[\lambda a'.\mu(T, x.B[T, x]; a'), a]$, where the definition of the type has been unrolled.

! **rule** `precMemberEquality`
{| intro_resource []**;** eqcd_resource **|}** H z:
[wf**]** [·] $H \vdash \mu\{T,x.B[T; x]; a\}$ Type $\longrightarrow$
[wf**]** [·] $H \vdash a1 = a2 \in B[(\lambda z.\mu\{T,x.B[T; x]; z\}); a] \longrightarrow$
[*ext*] $H \vdash a1 = a2 \in \mu\{T,x.B[T; x]; a\}$
= ·

Elimination

The elimination form performs induction on the recursive type definition. The conclusion $G[p]$ holds on any element p of the recursive type $\mu(T, x.B[T, x]; a)$ if, given that it holds on all the unrollings of p, it also holds on p.

! **rule** `precElimination`
{| elim_resource [(ThinOption thinT)] **|}**
[wf**]** [·] H; r: $\mu\{T,x.B[T;\,x];\,a\}$; $J[r]\vdash\ a \in A \longrightarrow$
$g[r;\,u;\,p;\,h]$**:** **[**main**]** [*ext*]
1. H
2. r: $\mu\{T,x.B[T;\,x];\,a\}$
3. $J[r]$
4. Z: $A \to \mathbb{U}$[i]
5. u: $(a{:}A \times (Z\ a)) \subseteq (a{:}A \times \mu\{T,x.B[T;\,x];\,a\})$
6. h: p:$(a{:}A \times (Z\ a)) \to G[p]$
7. p: $a{:}A \times B[Z;\,a]$
$\vdash G[p] \longrightarrow$

[*ext*] H; r: $\mu\{T,x.B[T;\,x];\,a\}$; $J[r]\vdash\ G[a]$
= precind$(a;\,p,p.\ g[r;\,(\lambda x.\cdot);\,p;\,h]$

The second form of elimination performs an unrolling of the type definition of the parameterized recursive type.

! **rule** `precUnrollElimination`
{| elim_resource [(ThinOption thinT)] **|}** H J z y u:
$g[z;\,y;\,u]$**:** [*ext*]
1. H
2. r: $\mu\{T,x.B[T;\,x];\,a\}$
3. $J[r]$
4. y: $B[\lambda z.\mu\{T,x.B[T;\,x];\,z\};\,a]$
5. u: $r = y \in B[(\lambda z.\mu\{T,x.B[T;\,x];\,z\});\,a]$
$\vdash G[y] \longrightarrow$

[*ext*] H; r: $\mu\{T,x.B[T;\,x];\,a\}$; $J[r]\vdash\ G[r]$
= $g[z;\,z;\,\cdot]$

Combinator equality

The `precind` term $prec_ind(r; h, z.t[h,z])$ produces values of type S if the argument r is the pair of a parameter a and a term in some parameterized recursive type $\mu(T, y.B[T,y]; a)$, and the body $t[h,z]$ produces values of type S given the argument r, and a function h that computes the result on the unrollings.

! **rule** `precindEquality`
{| intro_resource [**];** eqcd_resource **|}**
H $(\lambda x.S[x])$ $(a{:}A \times \mu\{T,y.B[T;$
[wf**]** [·]
1. H

2. Z: $A \to \mathbb{U}$[i]
3. u: $(a{:}A \times (Z\ a)) \subseteq (a{:}A \times \mu\{T,x.B[T;\ x];\ a\})$
4. h: z:$(a{:}A \times (Z\ a)) \to S[z]$
5. z: $a{:}A \times B[Z;\ a]$
$\vdash\ t1[h;\ z] = t2[h;\ z] \in S[z] \longrightarrow$

[*ext*]
1. H
$\vdash$
precind($r1$; $h1$,$h1$. $t1$[$h1$; $z1$]
= precind($r2$; $h2$,$h2$. $t2$[$h2$; $z2$] $\in S[r1]$
= ·

14.27 Itt_srec module

The `Itt_srec` module defines a "simple" recursive type, without parameters that are passed along the unrollings of the type, as it is in the parameterized recursive type in `Itt_prec`.

The syntax of the recursive type is $\mu(T.B[T])$. The variable T represents the type itself, which is given through the interpretation $T = B[T]$. The body $B[T]$ must be a type for *any* type $T \in \mathbb{U}_i$, and in addition $B[T]$ must be monotone in the type argument T.

14.27.1 Parents

include Itt_equal
include Itt_prec
include Itt_subtype
include Itt_void
include Itt_struct

14.27.2 Terms

The `srec` term defines the recursive type. The `srecind` term defines an induction combinator over elements of the recursive type.

declare `srec` = Itt_srec!srec{T. B['T]}
declare `srecind` = Itt_srec!srecind{'a; p, h. g['p; 'h]}

14.27.3 Rewrites

The `srecind` induction combinator takes an argument a that belongs to a recursive type definition. The computation is defined through the body $g[p, h]$, which takes a recursive instance p, and the argument element h.

! **rewrite** `unfold_srecind` :
srecind(a; p,p. $g[p; h]$ $\longleftrightarrow$
$g[(\lambda a.\text{srecind}(a; p,p.\ g[p; h]); a]$

14.27.4 Rules

Typehood and equality

The simple recursive type $\mu(T.B[T])$ is a type if $B[T]$ is a monotone type over types type $T \in \mathbb{U}_i$.

! **rule** `srecEquality`
{| intro_resource []**;** eqcd_resource **|}** H T S1 S2 z:
[wf**]** [·] H; T: $\mathbb{U}$[i]$\vdash$ $B1[T] = B2[T] \in \mathbb{U}$[i] $\longrightarrow$
[wf**]** [·] H; S1: $\mathbb{U}$[i]; S2: $\mathbb{U}$[i]; z: $S1 \subseteq S2 \vdash$ $B1[S1] \subseteq B1[S2]$ $\longrightarrow$
[*ext*] $H \vdash$ $\mu\{T1.B1[T1]\} = \mu\{T2.B2[T2]\} \in \mathbb{U}$[i]
= ·
! **rule** `srecType` **{|** intro_resource [] **|}** H S1 S2 z $\mathbb{U}$[i]:
[wf**]** [·] H; S1: $\mathbb{U}$[i]; S2: $\mathbb{U}$[i]; z: $S1 \subseteq S2 \vdash$ $B[S1] \subseteq B[S2]$ $\longrightarrow$
[*ext*] $H \vdash$ $\mu\{T.B[T]\}$ Type
= ·

Membership

The elements of the recursive type $\mu(T.B[T])$ are the elements of $B[\mu(T.B[T])]$.

! **rule** `srec_memberEquality` **{|** intro_resource [] **|}** H:
[wf**]** [·] $H \vdash$ $x1 = x2 \in B[\mu\{T.B[T]\}]$ $\longrightarrow$
[wf**]** [·] $H \vdash$ $\mu\{T.B[T]\}$ Type $\longrightarrow$
[*ext*] $H \vdash$ $x1 = x2 \in \mu\{T.B[T]\}$
= ·

Elimination

The elimination form performs induction over the assumption $x{:}\mu(T.B[T])$. The conclusion $C[x]$ is true for the arbitrary element x, if, given that it holds on the unrollings, it also holds on x.

! **rule** `srecElimination`
{| elim_resource [(ThinOption thinT)] **|}**
$g[x;\ T1;\ u;\ w;\ z]$**:** **[**main**]** [*ext*]
1. H
2. x: $\mu\{T.B[T]\}$
3. $J[x]$
4. T1: $\mathbb{U}$[i]
5. u: $T1 \subseteq \mu\{T.B[T]\}$
6. w: v:$T1 \to C[v]$
7. z: $B[T1]$

$\vdash C[z] \longrightarrow$

[*ext*] H; x: $\mu\{T.B[T]\}$; $J[x] \vdash C[x]$
= srecind(x; p,p. $g[x;\ \mu\{T.B[T]\};\ \cdot;\ p;\ h]$

The second elimination form performs unrolling of the recursive type definition.

! **rule** `srecUnrollElimination`
{| elim_resource [(ThinOption thinT)] **|}** H J x y u:
$g[x;\ y;\ u]$**:** **[**main**]** [*ext*]
1. H
2. x: $\mu\{T.B[T]\}$
3. $J[x]$
4. y: $B[\mu\{T.B[T]\}]$
5. u: $x = y \in B[\mu\{T.B[T]\}]$

$\vdash C[y] \longrightarrow$

[*ext*] H; x: $\mu\{T.B[T]\}$; $J[x] \vdash C[x]$
= $g[x;\ x;\ \cdot]$

Combinator equality

The `srecind` term produces a value of type S if the argument belongs to some recursive type, and the body computes a value of type S given the argument r and a function h to compute the values of the recursive calls.

! **rule** `srecindEquality`
{| intro_resource []**;** eqcd_resource **|}**
[wf**]** [$\cdot$] $H \vdash r1 = r2 \in \mu\{T.B[T]\} \longrightarrow$

[wf**]** [·]
1. H
2. T1: $\mathbb{U}$[i]
3. z: $T1 \subseteq \mu\{T.B[T]\}$
4. v: w:$T1 \rightarrow S[w]$
5. w: $B[T1]$

$\vdash t1[v; w] = t2[v; w] \in S[w] \longrightarrow$
[*ext*]
1. H

$\vdash$
srecind($r1$; $h1$,$h1$. $t1[h1; z1]$
= srecind($r2$; $h2$,$h2$. $t2[h2; z2] \in S[r1]$
= ·

14.28 Itt_list module

The `Itt_list` module defines the type of finite lists of elements. The lists can be defined using the simple recursive type in module `Itt_srec`. However, the lists have a simpler semantics, and they are defined them as primitive, so that lists can be used without including the recursive type.

14.28.1 Parents

include Itt_equal
include Itt_rfun
include Itt_struct
include Itt_logic

14.28.2 Terms

The `nil` term is the empty list, the `cons` term adds an element a to list b.

declare `nil` = Itt_list!nil
declare `cons` = Itt_list!cons{'a; 'b}

The `list` term defines the list type. The `list_ind` term defines the induction combinator.

declare `list` = Itt_list!list{'a}
declare `list_ind`
= Itt_list!list_ind{'e; 'base; h, t, f. step['h; 't; 'f]}

14.28.3 Rewrites

The `list_ind` term computes values on lists. The combinator has two bodies; the *base* term defines the value on empty lists, and the $step[h, t, f]$ term defines values on $cons(h, t)$, where f represents the value computed on the tail t of the list.

! **rewrite** `reduce_listindNil` :
(match [] with [] -> *base* | h::t.f -> $step[h; t; f]$) ⟷
base

! **rewrite** `reduce_listindCons` :
(match u::v with [] -> *base* | h::t.f -> $step[h; t; f]$) ⟷
$step[u;$
v; (match v with [] -> *base* | h::t.f -> $step[h; t; f]$)]

14.28.4 Rules

Typehood and equality

The $list(T)$ term is a well-formed type if T is a type.

! **rule** `listType` {| intro_resource [] |} H:
[wf**]** [·] $H \vdash A$ Type ⟶
[*ext*] $H \vdash A$ List Type
= ·
! **rule** `listEquality` {| intro_resource [] |} H:
[wf**]** [·] $H \vdash A = B \in \mathbb{U}$[i] ⟶
[*ext*] $H \vdash A$ List = B List $\in \mathbb{U}$[i]
= ·
! **rule** `nilFormation` {| intro_resource [] |} H:
[wf**]** [·] $H \vdash A$ Type ⟶
[*ext*] $H \vdash A$ List
= []

Membership

The `nil` term is a member of every list type $list(A)$.

! **rule** `nilEquality` **{|** intro_resource [] **|}** H:
[wf**]** [·] $H \vdash A$ Type ⟶
[*ext*] $H \vdash [] \in A$ List
= ·

The `cons` term $cons(h, t)$ is a member of the list type $list(A)$ if h is an element of A, and t is an element of $list(A)$.

! **rule** `consEquality` **{|** intro_resource []**;** eqcd_resource **|}** H:
[wf**]** [·] $H \vdash u1 = u2 \in A$ ⟶
[wf**]** [·] $H \vdash v1 = v2 \in A$ List ⟶
[*ext*] $H \vdash u1::v1 = u2::v2 \in A$ List
= ·

Elimination

The elimination for performs induction over the assumption l: $list(A)$. The rule produces two cases for a conclusion $C[l]$. In the base case, C must hold on the empty list, and in the induction step, $C[cons(h, t)]$ must hold for any elements $h \in A$ and $t \in list(A)$, where the induction hypothesis $C[t]$ holds on t.

! **rule** `listElimination`
{| elim_resource [(ThinOption thinT)] **|}** H J l w u v:
base[l]**: [**main**]** [*ext*] H; l: A List; $J[l] \vdash C[[]]$ ⟶
step[l; u; v; w]**: [**main**]** [*ext*]
1. H
2. l: A List
3. $J[l]$
4. u: A
5. v: A List
6. w: $C[v]$
$\vdash C[u::v]$ ⟶
[*ext*] H; l: A List; $J[l] \vdash C[l]$
= (match l with [] -> *base*[l] | $u::v.w$ -> *step*[l; u; v; w])

Combinator equality

The `list_ind` term **match** l **with** $cons(v, z).base \to step[u, v, z]$ computes a value of type T if 1) the argument l is a list of type $list(A)$, 2) the *base* term has type T, and 3) the *step* term computes a value of type T for any elements $u \in A$, $v \in list(A)$, and $z \in T$.

! **rule** `list_indEquality`

{| intro_resource []**;** eqcd_resource **|}** H (λ*l*.*T*[*l*]) (*A* List) u v w:
[wf**]** [·] *H*⊢ *e1* = *e2* ∈ *A* List ⟶
[wf**]** [·] *H*⊢ *base1* = *base2* ∈ *T*[[]] ⟶
[wf**]** [·]
1. *H*
2. u: *A*
3. v: *A* List
4. w: *T*[*v*]
⊢ *step1*[*u*; *v*; *w*] = *step2*[*u*; *v*; *w*] ∈ *T*[*u*::*v*] ⟶
[*ext*]
1. *H*
⊢
match *e1* with [] -> *base1* | *u1*::*v1*.*z1* -> *step1*[*u1*; *v1*; *z1*]
= match *e2* with
[] ->
base2
| *u2*::*v2*.*z2* ->
step2[*u2*; *v2*; *z2*] ∈ *T*[*e1*]
= ·

Computation

The *only* representative on the empty list is the `nil` term.

! **rule** `nilSqequal` H *T*:
[·] *H*⊢ *u* = [] ∈ *T* List ⟶
[*ext*] *H*⊢ *u* ⟷ []
= ·

Subtyping

The list type *list*(*A*) is covariant in the type argument *A*.

[8,77]* **rule `listSubtype` H:
[·] *H*⊢ *A1* ⊆ *A2* ⟶
[*ext*] *H*⊢ (*A1* List) ⊆ (*A2* List)

Contradiction

The terms `nil` and `cons` are distinct in every list type.

[7,62]* **rule `nil_neq_cons` **{|** elim_resource [] **|}** H J:
[*ext*] *H*; x: [] = *h*::*t* ∈ *T* List; *J*[*x*]⊢ *C*[*x*]
[7,65]* **rule `cons_neq_nil` **{|** elim_resource [] **|}** H J:
[*ext*] *H*; x: *h*::*t* = [] ∈ *T* List; *J*[*x*]⊢ *C*[*x*]

14.29 Itt_list2 module

The `Itt_list2` module defines a "library" of additional operations on the lists defined in the `Itt_list` module.

14.29.1 Parents

include Itt_list
include Itt_logic
include Itt_bool
include Itt_int
include Itt_int_bool

14.29.2 Terms

The `is_nil` term defines a Boolean value that is true iff the argument list l is empty.

declare `is_nil` = Itt_list2!is_nil{'l}

The `append` term takes two lists and concatenates them.

declare `append` = Itt_list2!append{'l1; 'l2}

The `ball2` term defines a Boolean universal quantification over two lists. The test $b[x, y]$ must compute a Boolean value given elements of the two lists, and the test is *ff* if the lists have different lengths.

declare `ball2` = Itt_list2!ball2{'l1; 'l2; x, y. b['x; 'y]}

The `assoc` term defines an associative lookup on the list l. The list l should be a list of pairs. The `assoc` term searches for the element x as the first element of one of the pairs. On the first occurrence of a pair (x, y), the value $b[y]$ is returned. The z term is returned if a pair is not found.

declare `assoc` = Itt_list2!assoc{'eq; 'x; 'l; y. b['y]; 'z}

The `rev_assoc` term also performs an associative search, but it keys on the second element of each pair.

declare `rev_assoc`
= Itt_list2!rev_assoc{'eq; 'x; 'l; y. b['y]; 'z}

The `map` term applies the function f to each element of the list l, and returns the list of the results (in the same order).

declare `map` = Itt_list2!map{'f; 'l}

The `fold_left` term applies the function f to each element of the list l, together with an extra argument v. The result of each computation is passed as the argument v to the next stage of the computation.

declare `fold_left` = Itt_list2!fold_left{'f; 'v; 'l}

The `nth` term returns element i of list l. The argument i must be within the bounds of the list.

declare `nth` = Itt_list2!nth{'l; 'i}

The `replace_nth` term replace element i of list l with the term v.

declare `replace_nth` = Itt_list2!replace_nth{'l; 'i; 'v}

The `length` term returns the total number of elements in the list l.

declare `length` = Itt_list2!length{'l}

The `rev` function returns a list with the same elements as list l, but in reverse order.

declare `rev` = Itt_list2!rev{'l}

14.29.3 Rewrites

The `is_nil` term is defined with the `list_ind` term, with a base case *tt*, and step case *ff*.

! **rewrite** `unfold_is_nil` :
  (l =*b* []) ⟷
  (match l with [] -> true | h::t.g -> false)

[1,22]* **rewrite `reduce_is_nil_nil` : ([] =*b* []) ⟷ true
[1,22]* **rewrite `reduce_is_nil_cons` : (h::t =*b* []) ⟷ false

The `append` term performs induction over the first list.

! **rewrite** `unfold_append` :
  ($l1$ @ $l2$) ⟷
  (match $l1$ with [] -> $l2$ | h::t.g -> h::g)

[1,20]* **rewrite `reduce_append_nil` : ([] @ $l2$) ⟷ $l2$
[1,62]* **rewrite `reduce_append_cons` : (x::$l1$ @ $l2$) ⟷ x::($l1$ @ $l2$)

The `ball2` term performs simultaneous induction over both lists, comparing the elements of the lists with the comparison $b[x, y]$.

```
! rewrite unfold_ball2 :
    (∀bx, y ∈ l1, l2. b[x; y]) ⟷
    (match l1 with
           [] ->
         (λz.match z with   [] -> true | h::t.g -> false)
       | h1::t1.g1 ->
         (λz.match z with
                 [] ->
               false
             | h2::t2.g2 ->
               (b[h1; h2] ∧b (g1 t2))) l2)

*[1,38] rewrite reduce_ball2_nil_nil :
    (∀bx, y ∈ [], []. b[x; y]) ⟷
    true
*[1,42] rewrite reduce_ball2_nil_cons :
    (∀bx, y ∈ [], h::t. b[x; y]) ⟷
    false
*[1,38] rewrite reduce_ball2_cons_nil :
    (∀bx, y ∈ h::t, []. b[x; y]) ⟷
    false
*[1,272] rewrite reduce_ball2_cons_cons :
    (∀bx, y ∈ h1::t1, h2::t2. b[x; y]) ⟷
    (b[h1; h2] ∧b (∀bx, y ∈ t1, t2. b[x; y]))
```

The `assoc` term performs induction over the list, splitting each pair and comparing it the the key.

```
! rewrite unfold_assoc :
    (try let y = assoc x ∈eq l in b[y] with Not_found -> z) ⟷
    (match l with
           [] ->
         z
       | h::t.g ->
         (match h with (u,v) → (if (eq u x) then b[v] else g)))

*[1,22] rewrite reduce_assoc_nil :
    (try let y = assoc x ∈eq [] in b[y] with Not_found -> z) ⟷
    z
*[1,196] rewrite reduce_assoc_cons :
    (try let y = assoc x ∈eq (u,v)::l in b[y] with Not_found -> z) ⟷
```

```
(if (eq u x) then
   b[v]
 else
   (try let y = assoc x ∈eq l in b[y] with Not_found -> z))
```

The `rev_assoc` term also performs induction over the list, but it keys on the second element of each pair.

```
! rewrite unfold_rev_assoc :
   (try let y = rev_assoc x ∈eq l in b[y] with Not_found -> z) ⟷
   (match l with
         [] ->
       z
     | h::t.g ->
       (match h with (u,v) → (if (eq v x) then b[u] else g)))

*[1,22] rewrite reduce_rev_assoc_nil :
   (try let y = rev_assoc x ∈eq [] in b[y] with Not_found -> z) ⟷
   z
*[1,196] rewrite reduce_rev_assoc_cons :
   (try
      let y = rev_assoc x ∈eq (u,v)::l
      in
        b[y]
    with Not_found ->
      z) ⟷
   (if (eq v x) then
      b[u]
    else
      (try let y = rev_assoc x ∈eq l in b[y] with Not_found -> z))
```

The `map` term performs induction over the list *l*, applying the function to each element, and collecting the results.

```
! rewrite unfold_map :
   (map f l) ⟷
   (match l with   [] -> [] | h::t.g -> (f h)::g)

*[1,20] rewrite reduce_map_nil : (map f []) ⟷ []
*[1,86] rewrite reduce_map_cons : (map f h::t) ⟷ (f h)::(map f t)
```

The `fold_left` term performs induction over the list argument, applying the function to the head element and the argument computed by the previous stage of the computation.

! **rewrite** `unfold_fold_left` :
fold_left(f, v, l) ⟷
(match l with [] -> (λv.v) | h::t.g -> (λv.g (f h v)) v)

[1,30]* **rewrite `reduce_fold_left_nil` : fold_left(f, v, []) ⟷ v
[1,166]* **rewrite `reduce_fold_left_cons` :
fold_left(f, v, h::t) ⟷
fold_left(f, (f h v), t)

The `length` function counts the total number of elements in the list.

! **rewrite** `unfold_length` :
length(l) ⟷
(match l with [] -> 0 | u::v.g -> (g + 1))

[1,20]* **rewrite `reduce_length_nil` : length([]) ⟷ 0
[1,62]* **rewrite `reduce_length_cons` :
length(u::v) ⟷
(length(v) + 1)

The `nth` term performs induction over the list, comparing the index to 0 at each step and returning the head element if it reaches 0. The · term is returned if the index never reaches 0.

! **rewrite** `unfold_nth` :
nth(l, i) ⟷
(match l with
[] ->
·
| u::v.g ->
(λj.(if (j = 0) then u else (g (j - 1)))) i)

[2,167]* **rewrite `reduce_nth_cons` :
nth(u::v, i) ⟷
(if (i = 0) then u else nth(v, (i - 1)))

The `replace_nth` term is similar to the `nth` term, but it collects the list, and replaces the head element when the index reaches 0.

! **rewrite** `unfold_replace_nth` :
replace_nth(l, i, t) ⟷
(match l with
[] ->
[]
| u::v.g ->
(λj.(if (j = 0) then t::v else u::(g (j - 1)))) i)

[2,231]* **rewrite reduce_replace_nth_cons :
replace_nth(u::v, i, t) ⟷
(if (i = 0) then t::v else u::replace_nth(v, (i - 1), t))

The rev term reverses the list. This particular computation is rather inefficient; it appends the head of the list to the reversed tail.

! **rewrite** unfold_rev :
Itt_list2!rev{l} ⟷
(match l with [] -> [] | u::v.g -> (g @ [u]))

**rewrite** reduce_rev_nil : Itt_list2!rev{[]} ⟷ []
**rewrite** reduce_rev_cons :
Itt_list2!rev{u::v} ⟷
(Itt_list2!rev{v} @ [u])

14.29.4 Rules

The rules in the Itt_list2 are limited to well-formedness of each of the constructions.

[6,97]* **rule is_nil_wf **{|** intro_resource [] **|}** H T:
[wf**]** [·] H⊢ l ∈ T List ⟶
[*ext*] H⊢ l =b [] ∈ Bool
[12,141]* **rule append_wf2 **{|** intro_resource [] **|}** H:
[wf**]** [·] H⊢ $l1$ ∈ T List ⟶
[wf**]** [·] H⊢ $l2$ ∈ T List ⟶
[*ext*] H⊢ ($l1$ @ $l2$) ∈ T List
[8,386]* **rule ball2_wf2 **{|** intro_resource [] **|}** H $T1$ $T2$ u v:
[wf**]** [·] H⊢ $T1$ Type ⟶
[wf**]** [·] H⊢ $T2$ Type ⟶
[wf**]** [·] H⊢ $l1$ ∈ $T1$ List ⟶
[wf**]** [·] H⊢ $l2$ ∈ $T2$ List ⟶
[wf**]** [·] H; u: $T1$; v: $T2$⊢ b[u; v] ∈ Bool ⟶
[*ext*] H⊢ (∀$b$$x$, y ∈ $l1$, $l2$. b[x; y]) ∈ Bool
[9,365]* **rule assoc_wf **{|** intro_resource [] **|}** H z $T1$ $T2$:
[wf**]** [·] H⊢ $T2$ Type ⟶
[wf**]** [·] H⊢ eq ∈ ($T1$ → $T1$ → Bool) ⟶
[wf**]** [·] H⊢ x ∈ $T1$ ⟶
[wf**]** [·] H⊢ l ∈ ($T1$ × $T2$) List ⟶
[wf**]** [·] H; z: $T2$⊢ b[z] ∈ T ⟶
[wf**]** [·] H⊢ z ∈ T ⟶
[*ext*]

1. H
⊢ (try let v = assoc x ∈eq l in b[v] with Not_found -> z) ∈ T

[8,324]* **rule `rev_assoc_wf` {| intro_resource [] |} H z *T1 T2*:
[wf**]** [·] H⊢ *T1* Type ⟶
[wf**]** [·] H⊢ *eq* ∈ (*T2* → *T2* → Bool) ⟶
[wf**]** [·] H⊢ *x* ∈ *T2* ⟶
[wf**]** [·] H⊢ *l* ∈ (*T1* × *T2*) List ⟶
[wf**]** [·] H; z: *T1*⊢ *b*[*z*] ∈ *T* ⟶
[wf**]** [·] H⊢ *z* ∈ *T* ⟶
[*ext*]
1. H
⊢
(try let v = rev_assoc x ∈eq l in b[v] with Not_found -> z) ∈ T

[6,163]* **rule `map_wf` {| intro_resource [] |} H *T1*:
[wf**]** [·] H⊢ *T1* Type ⟶
[wf**]** [·] H⊢ *T2* Type ⟶
[wf**]** [·] H⊢ *f* ∈ (*T1* → *T2*) ⟶
[wf**]** [·] H⊢ *l* ∈ *T1* List ⟶
[*ext*] H⊢ map *f* *l* ∈ *T2* List

[7,248]* **rule `fold_left_wf` {| intro_resource [] |} H *T1 T2*:
[wf**]** [·] H⊢ *T1* Type ⟶
[wf**]** [·] H⊢ *T2* Type ⟶
[wf**]** [·] H⊢ *f* ∈ (*T1* → *T2* → *T2*) ⟶
[wf**]** [·] H⊢ *v* ∈ *T2* ⟶
[wf**]** [·] H⊢ *l* ∈ *T1* List ⟶
[*ext*] H⊢ fold_left(*f*, *v*, *l*) ∈ *T2*

[6,110]* **rule `length_wf` {| intro_resource [] |} H *T1*:
[wf**]** [·] H⊢ *T1* Type ⟶
[wf**]** [·] H⊢ *l* ∈ *T1* List ⟶
[*ext*] H⊢ length(*l*) ∈ ℤ

#[30,617] **rule** `nth_wf` {| intro_resource [] |} H:
[wf**]** [·] H⊢ *T* Type ⟶
[wf**]** [·] H⊢ *l* ∈ *T* List ⟶
[wf**]** [·] H⊢ *i*≥0 ⟶
[wf**]** [·] H⊢ *i* < length(*l*) ⟶
[*ext*] H⊢ nth(*l*, *i*) ∈ *T*

#[0,1] **rule** `replace_nth_wf` {| intro_resource [] |} H:
[wf**]** [·] H⊢ *T* Type ⟶
[wf**]** [·] H⊢ *l* ∈ *T* List ⟶
[wf**]** [·] H⊢ *i*≥0 ⟶
[wf**]** [·] H⊢ *i* < length(*l*) ⟶
[wf**]** [·] H⊢ *t* ∈ *T* ⟶
[*ext*] H⊢ replace_nth(*l*, *i*, *t*) ∈ *T* List

rule rev_wf {| intro_resource [] |} H:
[wf**]** [·] $H \vdash A$ Type $\longrightarrow$
[wf**]** [·] $H \vdash l \in A$ List $\longrightarrow$
[*ext*] $H \vdash$ Itt_list2!rev$\{l\} \in A$ List

14.30 Itt_quotient module

The `Itt_quotient` module defines the *quotient* type $x, y: T//E[x, y]$, which imposes a *new* equality E on the type T. The relation E must be an equivalence relation on T; and to be well-formed the new equality must be *coarser* than the native equality in T. The elements of the quotient type are the elements of T, but equality is determined by the relation E.

One use of the quotient type is to quotient a type in a similar manner as in set-theory. For example, the following type defines the integers *mod* 2.

$$\mathbb{Z}_2 \equiv i, j: \mathbb{Z}//i \textbf{ rem } 2 = j \textbf{ rem } 2$$

The even integers are equal in $\mathbb{Z}_2$, and so are the odd integers. Unlike set theory, the elements of the quotient type are not equivalence classes. They are the original elements; only the equality has changed.

Another use of the quotient is for *abstraction*. We could, for example, define data type of finite sets of numbers as follows:

$$\begin{array}{rcll} Set & \equiv & T & : \mathbb{U}_i \\ & \times & member & : T \to \mathbb{Z} \to \mathbb{U}_i \\ & \times & empty & : T \\ & \times & add & : T \to \mathbb{Z} \to T \\ & \times & empty_{axiom} & : \forall i: \mathbb{Z}. \neg(member(empty, i)) \\ & \times & add_{axiom} & : \forall t: T. \forall i, j: \mathbb{Z}. member(add(t, j), i) \Leftrightarrow \\ & & & \quad member(t, i) \vee i = j \end{array}$$

The data type definition, as-is, allows the type T to "escape" — the type T is just another field in the data type. Abstractly, we are usually only concerned with the membership function — two sets are equal if they have the same elements. If the implementation T is exposed, it is possible to construct functions that "peek" into the structure, possible producing non-functional behavior (with respect to the membership function).

One way to address this problem is to use the quotient type to "hide" the implementation. The first step is to pair the module implementation

with a set representative.

$$set \equiv S{:}\, Set \times S.T$$

The next step is to *quotient* the construction by its membership.

$$Set' \equiv S_1, S_2{:}\, set\,//\,\forall i\mathbb{Z}.S_1.1.member(S_1.2, i) \Leftrightarrow S_2.1.member(S_2.2, i)$$

Two sets in Set' are equal if-and-only-if they have the same membership. The other methods can be given wrapped definitions as follows:

$$\begin{array}{lcl} empty'(S) & \equiv & (S, S.empty) \\ member'(S, i) & \equiv & S.1.member(S.2, i) \\ add'(S, i) & \equiv & (S.1, S.1.add(S.2, i)) \end{array}$$

The types are as follows:

$$\begin{array}{lcl} empty' & : & Set \rightarrow Set' \\ member' & : & Set' \rightarrow \mathbb{Z} \rightarrow \mathbb{U}_i \\ add' & : & Set' \rightarrow \mathbb{Z} \rightarrow Set' \end{array}$$

In addition, the membership axioms `empty_axiom` and `add_axiom` can be proved for the Set' definition.

14.30.1 Parents

include Itt_equal
include Itt_set
include Itt_rfun
include Itt_struct

14.30.2 Terms

The `quot` type defines the quotient type $x, y{:}\, A//E[x, y]$.

declare `quot` = Itt_quotient!quot{'A; x, y. E['x; 'y]}

14.30.3 Rules

Equality and well-formedness

The quotient $x, y{:}\, A//E[x, y]$ if A is a type, and E is an *equivalence relation*:

- E is reflexive: $E[x, x]$
- E is symmetric: $E[x, y] \Rightarrow E[y, x]$
- E is transitive: $E[x, y] \Rightarrow E[y, z] \Rightarrow E[x, z]$

! **rule** `quotientType` **{|** intro_resource [] **|}** H u v w x1 x2:
[wf**]** [·] $H \vdash A$ Type ⟶
[wf**]** [·] H; u: A; v: $A \vdash E[u; v]$ Type ⟶
[wf**]** [·] H; u: $A \vdash E[u; u]$ ⟶
[wf**]** [·] H; u: A; v: A; x1: $E[u; v] \vdash E[v; u]$ ⟶
[wf**]** [·] H; u: A; v: A; w: A; x1: $E[u; v]$; x2: $E[v; w] \vdash E[u; w]$ ⟶
[*ext*] $H \vdash (x, y: A \,//\, E[x; y])$ Type
= ·

In the *weak* form, two quotient types $x, y: A_1 // E_1[x, y]$ and $x, y: A_2 // E_2[x, y]$ are equal if the types A_1 and A_2 and the equivalence relations E_1 and E_2 are equal.

! **rule** `quotientWeakEquality`
{| intro_resource []; eqcd_resource **|}** H x y z u v:
[wf**]** [·] $H \vdash A1 = A2 \in \mathbb{U}$[i] ⟶
[wf**]** [·] H; x: $A1$; y: $A1 \vdash E1[x; y] = E2[x; y] \in \mathbb{U}$[i] ⟶
[wf**]** [·] H; x: $A1 \vdash E1[x; x]$ ⟶
[wf**]** [·] H; x: $A1$; y: $A1$; u: $E1[x; y] \vdash E1[y; x]$ ⟶
[wf**]** [·]
1. H
2. x: $A1$
3. y: $A1$
4. z: $A1$
5. u: $E1[x; y]$
6. v: $E1[y; z]$
$\vdash E1[x; z]$ ⟶
[*ext*]
1. H
$\vdash$
$(x1, y1: A1 \,//\, E1[x1; y1]) = (x2, y2: A2 \,//\, E2[x2; y2]) \in \mathbb{U}$[i]
= ·

In the *strong* form, two quotient types $x, y: A_1 // E_1[x, y]$ and $x, y: A_2 // E_2[x, y]$ are equal if the types A_1 and A_2 are equal, and their equivalence relations are *extensionally* equal $E_1[x, y] \Leftrightarrow E_2[x, y]$.

! **rule** `quotientEquality`
{| intro_resource [(SelectOption 1)] **|}** H r s v:

[wf] [·] $H \vdash (x1, y1: A1 \mathbin{//} E1[x1; y1]) \in \mathbb{U}[i] \longrightarrow$
[wf] [·] $H \vdash (x2, y2: A2 \mathbin{//} E2[x2; y2]) \in \mathbb{U}[i] \longrightarrow$
[wf] [·] $H \vdash A1 = A2 \in \mathbb{U}[i] \longrightarrow$
[wf] [·] H; v: $A1 = A2 \in \mathbb{U}[i]$; r: $A1$; s: $A1 \vdash E1[r; s] \to E2[r; s] \longrightarrow$
[wf] [·] H; v: $A1 = A2 \in \mathbb{U}[i]$; r: $A1$; s: $A1 \vdash E2[r; s] \to E1[r; s] \longrightarrow$
[*ext*]
1. H
$\vdash$
$(x1, y1: A1 \mathbin{//} E1[x1; y1]) = (x2, y2: A2 \mathbin{//} E2[x2; y2]) \in \mathbb{U}[i]$
$= \cdot$

Membership

In the *weak* form, any two elements in A are also in the quotient $x, y: A//E[x, y]$ for *any* equivalence relation E.

! **rule** `quotient_memberWeakEquality` {| intro_resource [] |} H:
[wf] [·] $H \vdash (x, y: A \mathbin{//} E[x; y])$ Type $\longrightarrow$
[wf] [·] $H \vdash a1 = a2 \in A \longrightarrow$
[*ext*] $H \vdash a1 = a2 \in (x, y: A \mathbin{//} E[x; y])$
$= \cdot$

! **rule** `quotientMember` {| intro_resource [] |} H:
[wf] [·] $H \vdash (x, y: A \mathbin{//} E[x; y])$ Type $\longrightarrow$
[wf] [·] $H \vdash l \in A \longrightarrow$
[*ext*] $H \vdash l \in (x, y: A \mathbin{//} E[x; y])$
$= \cdot$

In the *strong* form, two elements a_1 and a_2 are in the quotient type $x, y: A//E[x, y]$ if they are equal in A, and they are related with the equivalence relation $E[a_1, a_2]$.

! **rule** `quotient_memberEquality`
{| intro_resource []; eqcd_resource |} H:
[wf] [·] $H \vdash (x, y: A \mathbin{//} E[x; y])$ Type $\longrightarrow$
[wf] [·] $H \vdash a1 \in A \longrightarrow$
[wf] [·] $H \vdash a2 \in A \longrightarrow$
[wf] [·] $H \vdash E[a1; a2] \longrightarrow$
[*ext*] $H \vdash a1 = a2 \in (x, y: A \mathbin{//} E[x; y])$
$= \cdot$

Elimination

The first two elimination forms are valid only if the goal is an equality judgment. For both cases, the judgment is true if it is true for any

two elements that are equal in the quotient type. The difference is the ordering of the hypotheses in the subgoal.

```
! rule quotientElimination1 H J v w z:
    [wf] [·] H; a: x, y: A // E[x; y]; J[a]⊢ T[a] Type ⟶
    [main] [·]
            1. H
            2. a: x, y: A // E[x; y]
            3. J[a]
            4. v: A
            5. w: A
            6. z: E[v; w]
            ⊢ s[v] = t[w] ∈ T[v] ⟶
    [ext] H; a: x, y: A // E[x; y]; J[a]⊢ s[a] = t[a] ∈ T[a]
    = ·
! rule quotientElimination2
    {| elim_resource [(ThinOption thinT)] |} H J v w z:
    [wf] [·] H; a: x, y: A // E[x; y]; J[a]⊢ T[a] Type ⟶
    [main] [·]
            1. H
            2. a: x, y: A // E[x; y]
            3. v: A
            4. w: A
            5. z: E[v; w]
            6. J[v]
            ⊢ s[v] = t[w] ∈ T[v] ⟶
    [ext] H; a: x, y: A // E[x; y]; J[a]⊢ s[a] = t[a] ∈ T[a]
    = ·
```

An equality assumption $a_1 = a_2 \in x, y{:}A//E[x, y]$ implies that $E[a_1, a_2]$.

```
! rule quotient_equalityElimination {| elim_resource [] |}
    g[v]: [main] [ext]
                1. H
                2. x: a1 = a2 ∈ (x, y: A // E[x; y])
                3. J[x]
                4. v: [E[a1; a2]]
                ⊢ T[x] ⟶
    [ext] H; x: a1 = a2 ∈ (x, y: A // E[x; y]); J[x]⊢ T[x]
    = g[·]
```

Subtyping

The quotient $x, y: A//E[x, y]$ is covariant in the type A and the the equivalence relation E (the relation must become coarser).

! **rule** `quotientSubtype` H a1 a2:
[·] $H \vdash A1 \subseteq A2 \longrightarrow$
[·] H; a1: $A1$; a2: $A1 \vdash E2[a1; a2] \longrightarrow$
[·] $H \vdash (x1, y1: A1 // E1[x1; y1])$ Type $\longrightarrow$
[·] $H \vdash (x2, y2: A2 // E2[x2; y2])$ Type $\longrightarrow$
[*ext*]
1. H
$\vdash (x1, y1: A1 // E1[x1; y1]) \subseteq (x2, y2: A2 // E2[x2; y2])$
$= \cdot$

14.31 Itt_esquash module

In many cases it is convenient to "squash" (omit) the computational content of a proposition. The "set" type defined in the `Itt_set` module defines a squashed proposition $P[x]$ with the type $\{x: A \mid P[x]\}$. The elements of the set are the elements in $a \in A$ where the proposition $P[a]$ is true, but the computational content for $P[a]$ is omitted.

The `Itt_esquash` module defines a generic squash term $esquash(P)$. The elements of the type are the trivial terms $\cdot$, and two terms $esquash(P_1)$ and $esquash(P_2)$ have the *extensional* equality $P_1 \Leftrightarrow P_2$.

The definition of the `esquash` term is in terms of the quotient type `quot` in the `Itt_quotient` module and the Boolean type in the `Itt_bool` module. The definition may look strange:

$$esquash(P) \equiv tt = ff \in b_1, b_2: Bool//(b_1 = b_2 \in Bool) \vee P.$$

The quotient type $b_1, b_2: Bool//(b_1 = b_2 \in Bool) \vee P$ equates tt and ff if P is true. Otherwise, $tt \neq ff$ and the equality type is empty.

A simpler construction (and the traditional Nuprl construction) defines the squash type with the set type:

$$squash(P) \equiv \{x: Unit \mid P\}.$$

The elements of `esquash` and `squash` are the same, but the equalities differ, $esquash(P_1) = esquash(P_2) \in \mathbb{U}_i$ if $P_1 \Leftrightarrow P_2$, and $squash(P_1) = squash(P_2) \in \mathbb{U}_i$ if $P_1 = P_2 \in \mathbb{U}_i$. This is because the quotient type uses an *extensional* equality on it's equivalence relation, and the `set` type uses an *intensional* equality on it's predicate.

14.31.1 Parents

include Itt_void
include Itt_unit
include Itt_quotient
include Itt_set
include Itt_bool
include Itt_logic

14.31.2 Terms

The `esquash_bool` term defines the quotient type, and the `esquash` term defines the squash predicate P.

declare `esquash_bool` = Itt_esquash!esquash_bool{'P}
declare `esquash` = Itt_esquash!esquash{'P}

14.31.3 Rewrites

The `esquash_bool` term defines the quotient predicate. The `esquash` term equates *tt* and *ff* in `esquash_bool`.

! **rewrite** `unfold_esquash_bool` :
esquash_bool(P) $\longleftrightarrow$
($b1$, $b2$: Bool // (($b1$ = $b2$ $\in$ Bool) $\vee$ P))

! **rewrite** `unfold_esquash` : ($\downarrow P$) $\longleftrightarrow$
(true = false $\in$ esquash_bool(P))

14.31.4 Rules

Typehood and equality

The `esquash_bool` term *esquash_bool*(P) is well-formed if P is a type; it contains the terms *tt* and *ff*.

[12,226]* **rule `esquash_bool_type` **{|** intro_resource [] **|}** H:
[wf**]** [$\cdot$] $H \vdash$ P Type $\longrightarrow$
[*ext*] $H \vdash$ esquash_bool(P) Type
[14,201]* **rule `esquash_bool_univ` **{|** intro_resource [] **|}** H:
[wf**]** [$\cdot$] $H \vdash$ $P \in \mathbb{U}$[i] $\longrightarrow$

[*ext*] $H \vdash$ esquash_bool(P) $\in \mathbb{U}$[i]
*[13,252] **rule** esquash_bool_true {| intro_resource [] |} H:
[wf**]** [·] $H \vdash P$ Type $\longrightarrow$
[*ext*] $H \vdash$ true $\in$ esquash_bool(P)
*[13,257] **rule** esquash_bool_false {| intro_resource [] |} H:
[wf**]** [·] $H \vdash P$ Type $\longrightarrow$
[*ext*] $H \vdash$ false $\in$ esquash_bool(P)

The esquash term inhabits the type universe $\mathbb{U}_i$ if the proposition P is also in $\mathbb{U}_i$.

*[25,515] **rule** esquash_type {| intro_resource [] |} H:
[wf**]** [·] $H \vdash P$ Type $\longrightarrow$
[*ext*] $H \vdash (\downarrow P)$ Type
*[3,44] **rule** esquash_univ {| intro_resource [] |} H:
[wf**]** [·] $H \vdash P \in \mathbb{U}$[i] $\longrightarrow$
[*ext*] $H \vdash \downarrow P \in \mathbb{U}$[i]

Introduction

The *esquash*(P) proposition is true if P is true. However, this rule is too strong to add to the intro_resource directly. Instead, the esquashT tactic is defined below to apply this rule.

*[15,229] **rule** esquash_intro H:
[main**]** [·] $H \vdash P \longrightarrow$
[*ext*] $H \vdash \downarrow P$

Elimination

The element in the *esquash*(P_1) term is always the term ·. The assumption x: *esquash*(P) can be "exposed" if the proof in not needed, as in the proof of *Void*. As the type theory evolves, it seems likely that we will allow the hypothesis to be exposed in squash sequent.

#[0,1] **rule** esquash_equal_elim H J:
[main**]** [·] H; x: $P1$; J[·]$\vdash$ $t1$[·] = $t2$[·] $\in$ $T3$[·] $\longrightarrow$
[*ext*] H; x: $\downarrow P1$; J[x]$\vdash$ $t1$[x] = $t2$[x] $\in$ $T3$[x]
#[0,1] **rule** esquash_void_elim H J:
[main**]** [·] H; x: $P1$; J[·]$\vdash$ Void $\longrightarrow$
[*ext*] H; x: $\downarrow P1$; J[x]$\vdash$ Void
#[0,1] **rule** esquash_esquash_elim H J:
[main**]** [·] H; x: $P1$; J[·]$\vdash$ $\downarrow P2$[·] $\longrightarrow$
[*ext*] H; x: $\downarrow P1$; J[x]$\vdash$ $\downarrow P2$[x]

The *esquash*(P) hypothesis can also be unhidden if P is an equality judgment, or *Void*, or *Unit*, or a nested `esquash` proposition (or in general if the proof can be recovered).

#[0,1] **rule** `esquash_equal_elim2` {| elim_resource [] |} H J:
[*ext*] H; x: $t1 = t2 \in T3$; $J[\cdot] \vdash C[\cdot] \longrightarrow$
[*ext*] H; x: $\downarrow(t1 = t2 \in T3)$; $J[x] \vdash C[x]$
#[0,1] **rule** `esquash_void_elim2` {| elim_resource [] |} H J:
[*ext*] H; x: $\downarrow$Void; $J[x] \vdash C[x]$
#[0,1] **rule** `esquash_unit_elim2` {| elim_resource [] |} H J:
[*ext*] H; $J[\cdot] \vdash C[\cdot] \longrightarrow$
[*ext*] H; x: $\downarrow$Unit; $J[x] \vdash C[x]$
#[0,1] **rule** `esquash_esquash_elim2` {| elim_resource [] |} H J:
[*ext*] H; x: $\downarrow P$; $J[\cdot] \vdash C[\cdot] \longrightarrow$
[*ext*] H; x: $\downarrow\downarrow P$; $J[x] \vdash C[x]$

Extensional equality

Two squashed propositions *esquash*(P_1) and *esquash*(P_2) are equal if both P_1 and P_2 are types, and $P_1 \Leftrightarrow P_2$.

#[0,1] **rule** `esquash_equal_intro` {| intro_resource [] |} H x:
[wf**]** [$\cdot$] $H \vdash P1 \in \mathbb{U}$[i] $\longrightarrow$
[wf**]** [$\cdot$] $H \vdash P2 \in \mathbb{U}$[i] $\longrightarrow$
[main**]** [$\cdot$] H; x: $P1 \vdash P2 \longrightarrow$
[main**]** [$\cdot$] H; x: $P2 \vdash P1 \longrightarrow$
[*ext*] $H \vdash \downarrow P1 = \downarrow P2 \in \mathbb{U}$[i]

Set elimination

The standard rule for set elimination `setElimination` can be overridden with the following more useful rule.

[5,31]* **rule `set_elim` {| elim_resource [] |} H J:
[main**]** [*ext*] H; x: T; z: $\downarrow P[x]$; $J[x] \vdash C[x] \longrightarrow$
[*ext*] H; x: $\{ y{:}T \mid P[y]\}$; $J[x] \vdash C[x]$

14.31.5 Tactics

`esquashT` The `esquashT` tactic applies the `esquash_intro` rule

14.32 Itt_theory module

The `Itt_theory` module collects all of the module in the Nuprl type theory. This is the basic module to use when stating and proving theorems the the Nuprl type theory.

14.32.1 Parents

include Base_theory
include Itt_equal
include Itt_void
include Itt_bool
include Itt_atom
include Itt_int
include Itt_rfun
include Itt_dfun
include Itt_fun
include Itt_well_founded
include Itt_dprod
include Itt_prod
include Itt_union
include Itt_struct
include Itt_set
include Itt_isect
include Itt_tunion
include Itt_bisect
include Itt_bunion
include Itt_subtype
include Itt_w
include Itt_prec
include Itt_srec
include Itt_list
include Itt_list2
include Itt_quotient
include Itt_esquash
include Itt_decidable

Chapter 15

Aczel's CZF Set Theory

The *CZF* theory formalizes Aczel's constructive set theory [Acz86]. In *Notes on Constructive Set Theory* [Acz97], Aczel provides the central motivation for the set theory.

> "The general topic of Constructive Set Theory originated in the seminal 1975 paper of John Myhill, where a specific axiom system CST was introduced. Constructive Set Theory provides a standard set theoretical framework for the development of constructive mathematics in the style of Errett Bishop [Bis67] and is one of several such frameworks for constructive mathematics that have been considered. It is distinctive in that it uses the standard first order language of classical axiomatic set theory and makes no explicit use of specifically constructive ideas. Of course its logic is intuitionistic, but there is no special notion of construction or constructive object. There are just the sets, as in classical set theory. This means that mathematics in constructive set theory can look very much like ordinary classical mathematics. The advantage of this is that the ideas, conventions and practise of the set theoretical presentation of ordinary mathematics can be used also in the set theoretical development of constructive mathematics, provided that a suitable discipline is adhered to. In the first place only the methods of logical reasoning available in intuitionistic logic should be used. In addition only the set theoretical axioms allowed in constructive set theory can be used. With some practise it is not difficult for the constructive mathematician to adhere to this discipline."

Aczel assures the constructivity of the set theory by using Martin–Löf's Constructive Set Theory as its foundation. We follow Aczel's ap-

proach in the definition of our formalization of the set theory. We encode the sets within the Nuprl type theory. There are two main reasons for formalizing *CZF*. First, we demonstrate the method for *deriving* one non-trivial theory from another (*CZF* is *derived* from the Nuprl type theory). In addition, set theory can often present a convenient framework for developing constructive mathematics using ordinary mathematical concepts. Aczel continues,

> "Why not present constructive mathematics directly in the type theory? This is an obvious option for the constructive mathematician. It has the drawback that there is no extensive tradition of presenting mathematics in a type theoretic setting. So, many techniques for representing mathematical ideas in a set theoretical language have to be reconsidered for a type theoretical language. This can be avoided by keeping to the set theoretical language."

The *CZF* set theory is defined in an intuitionistic first-order logic, with one additional primitive operator for set membership. All non-propositional elements of the set theory are sets; the numbers and other structures are coded in the usual manner. Sets us an *extensional* equality; two sets considered equal if they have the same elements.

15.1 Czf_itt_theory module

The *CZF* set theory is defined using a collection of axioms about set membership in a manner similar to the definition of *ZF* set theory. At the moment, the extent of our formalization is to encode the sets and prove that each of these axioms is valid given our interpretation of sets in the Nuprl type theory (defined in the `Itt_theory` module).

15.1.1 Axioms

Pairing

$$\forall a.\forall b.\exists c.\forall y.(y \in c \Leftrightarrow y = a \vee y = b)$$

Union

$$\forall a.\exists b.\forall y.(y \in b \Leftrightarrow \exists x \in a.(y \in x))$$

Infinity

$$\exists a.(\exists x.x \in a \wedge \forall x \in a.\exists y \in a.x \in y)$$

Extensionality

$$\forall a.\forall b.((\forall x.x \in a \Leftrightarrow x \in b) \Rightarrow a = b)$$

Set induction

$$\forall a.((\forall x \in a.\phi(x) \Rightarrow \phi(a)) \Rightarrow (\forall a.\phi(a)))$$

Restricted separation

The constructive version of separation is stated for an arbitrary *restricted* formula ϕ, where a formula is restricted is all quantifiers are restricted (either $\exists x \in y\ldots.$ or $\forall x \in y\ldots..$

$$\forall a.\exists b.\forall x.(x \in b \Leftrightarrow (x \in a \wedge \phi(x)))$$

Collection

The *CZF* set theory uses the *collection* schemes instead of the general replacement axiom, which would be defined as follows:

$$\forall a.((\forall x \in a.\exists! y.\phi(x,y)) \Rightarrow \exists b.\forall y.(y \in b \Leftrightarrow \exists x \in a.\phi(x,y))$$

The collection axioms are defined in terms of a predicate $\mathbb{B}$.

$$\mathbb{B}x \in a, y \in b.\phi \equiv (\forall x \in a.\exists y \in b.\phi) \wedge (\forall y \in b.\exists x \in a.\phi)$$

The *strong collection* scheme is defined as follows:

$$\forall a.(\forall x \in a.\exists y.\phi(x,y)) \Rightarrow (\exists b.\mathbb{B}x \in a, y \in b.\phi(x,y)).$$

The *subset collection* scheme defines a collection of subsets.

$$\forall a.\forall b.\exists c.(\forall u.\forall x \in a.\exists y \in b.\phi(x,y,u) \Rightarrow \exists z \in c.\mathbb{B}x \in a, y \in b.\phi(x,y,u))$$

15.1.2 Theory organization

For our formalization of the set theory, we present a rule-based approach more suitable to a presentation in a sequent calculus. We present the set theory in three parts. Fir we introduce the primitive propositions for membership and equality; we then characterize the first order logic; and the in the final section we present set *constructors* that correspond to each of the axioms of the set theory, together with well-formedness rules. The rules for the constructors are presented in the form of introduction and elimination rules for reasoning about membership in each of the set constructions. The following is a brief summary of the the account.

Encoding

The type of sets is defined in `Czf_itt_set` module as the W-type $set \equiv W(T{:}\mathbb{U}_1.T)$. That is, each set has an index type $T \in \mathbb{U}_1$, and a function $T \to set$ that lists the elements of the set.

Propositions

Equality $equal(s_1, s_2)$ The `Czf_itt_eq` module defines extensional set equality. The type-theoretic equality on the W-type is too fine for use in the set theory — two sets will differ if they have different index types, even though they may have the same elements. The `Czf_itt_set` module contains a discussion of the inadequacy of a quotient construction. Instead the `Czf_itt_eq` module defines several *functionality* judgments that are used to state well-formedness and functionality properties of set propositions and constructors.

Membership $s_1 \in_s s_2$ The `Czf_itt_member` module defines membership, and proves the theorem of *extensionality*: two sets are equal if they have the same elements.

Logic The modules in `Czf_itt_logic` define the first-order logic, including the restricted quantifiers $\forall x \in_s s.P[x]$ and $\exists x \in_s s.P[x]$, and the unrestricted quantifiers $\forall_s x.P[x]$ and $\forall_s x.P[x]$. The modules in the logic prove the functionality of the logical operators in addition to the $restricted(P)$ judgment for use in the separation construction.

Subset $s_1 \subseteq_s s_2$ The subset judgment $s_1 \subseteq_s s_2$ is defined in the `Czf_itt_subset` module as the proposition $\forall x \in_s s_1.x \in_s s_2$.

Constructors

Separation $\{x \in_s s \mid P[x]\}$ The `Czf_itt_sep` defines the constructor for restricted separation $\{x \in_s s \mid P[x]\}$, where s is a set, and $P[x]$ must be a *restricted* proposition for any set x. The definition of restriction is equivalent to Aczel's definition (that $P[x]$ contain no unrestricted quantifiers), but we give it a somewhat more general definition $\mathit{restricted}(P) \equiv P \in \mathbb{U}_1$. This definition is never exposed in the set theory; we provide rules for proving restriction for each of the restricted logical operators.

Empty set $\{\}$ The empty set is defined in the `Czf_itt_empty` module.

Singleton set $\{s\}$ The singleton $\{s\}$ set, defined in the `Czf_itt_singleton` module, defines a set containing one element, the set s.

Binary union $s_1 \cup s_2$**, general union** $\bigcup s$ The `Czf_itt_union` module defines two forms of union. The binary union $s_1 \cup s_2$ is the set containing the elements of s_1 and s_2. The general union $\bigcup s$ forms the set containing the elements of all the sets in s.

Pairing (s_1, s_2) The `Czf_itt_pair` module defines the unordered pair (s_1, s_2), which is a set containing the two elements s_1 and s_1.

Infinity (the natural numbers) ω The `Czf_itt_nat` module defines an infinite set ω that corresponds to the set of natural numbers. It also defines a zero element $0_s \in_s \omega$, and a successor $s(i)$, where $s(i) \in_s \omega$ if $i \in_s \omega$. We also formalize an induction principle over the natural numbers, as well as the total order $i <_s j$.

15.1.3 Parents

include Czf_itt_subset
include Czf_itt_power

15.2 Czf_itt_set module

The `Czf_itt_set` module provides the basic definition of sets and their elements. The `set` term denotes the type of all sets, defined using the W-type in the module `Itt_w`, as follows:

$$set \equiv W(T{:}\mathbb{U}_1.T).$$

That is, the *sets* are pairs of a type $T \in \mathbb{U}_1$, and a function $T \to set$ that specifies the elements of the set. Note that the type T can be *any* type in $\mathbb{U}_1$; equality of sets is is general undecidable, and their members can't necessarily be enumerated. This is a *constructive* theory, not a decidable theory. Of course, there will be special cases where equality of sets is decidable.

The sets are defined with the terms $collect(x \in T.f[x])$, where T is a type in $\mathbb{U}_1$, and $f[x]$ is a set for any index $x \in T$. The sets $f[x]$ are the *elements* of the set, and T is the a type used as their index. For example, the following set is empty.

$$\{\} = collect(x \in Void.x)$$

The following set is the singleton set containing the empty set.

$$\{\{\}\} = collect(x \in Unit.\{\})$$

The following set is equivalent.

$$\{\{\}\}' = collect(x \in \mathbb{Z}.\{\})$$

This raises an important point about equality. The membership equality defined ion W-types requires equality on the index type T as well as the element function f. The two sets $\{\{\}\}$ and $\{\{\}\}'$ are *not* equal in this type because they have the provably different index types *Unit* and $\mathbb{Z}$, even though they have the same elements.

One solution to this problem would be to use a quotient construction using the quotient type defined in the `Itt_quotient` module. The `Czf_itt_eq` module defines extensional set equality $\equiv_{ext}$, and we could potentially define the "real" sets with the following type definition.

$$real_sets \equiv (s_1, s_2{:}set\,//\,s_1 \equiv_{ext} s_2)$$

This type definition would require explicit functionality reasoning on all functions over *real_set*. This construction, however, makes these functionality proofs impossible because it omits the computational content of the equivalence judgment, which is a necessary part of the proof.

Alternative quotient formulations may be possible, but we have not pursued this direction extensively. Instead, we introduce set equality in the `Itt_set_eq` module, together with explicit functionality predicates for set operators. In addition, we prove the functionality properties for all the primitive set operations.

One avenue for improvement in this theory would be to stratify the

set types to arbitrary type universes $\mathbb{U}_i$, which would allow for higher-order reasoning on sets, classes, etc.

15.2.1 Parents

include Itt_theory

15.2.2 Terms

The `set` term defines the type of sets; the `collect` terms are the individual sets. The `isset` term is the well-formedness judgment for the *set* type. The `set_ind` term is the induction combinator for computation over sets. The *s* argument represents the set; *T* is it's index type, *f* is it's function value, and *g* is used to perform recursive computations on the children.

declare `set` = Czf_itt_set!set
declare `isset` = Czf_itt_set!isset{'s}
declare `collect` = Czf_itt_set!collect{'T; x. a['x]}
declare `set_ind` = Czf_itt_set!set_ind{'s; T, f, g. b['T; 'f; 'g]}

15.2.3 Rewrites

The following four rewrites give the primitive definitions of the set constructions from the terms in the Nuprl type theory.

! **rewrite** `unfold_set` : set $\longleftrightarrow$ ($\mathbb{W}x$:$\mathbb{U}$[1].x)
! **rewrite** `unfold_isset` : (s set) $\longleftrightarrow$ ($s \in$ set)
! **rewrite** `unfold_collect` :
(collect x: T. $a[x]$) $\longleftrightarrow$
tree(T, ($\lambda x.a[x]$))

! **rewrite** `unfold_set_ind` :
(let set(x, f).g = s in $b[x; f; g]$) $\longleftrightarrow$
(tree_ind(g. let tree(x, f) = s in $b[x; f; g]$)

The `set_ind` term performs a pattern match; the normal reduction sequence can be derived from the computational behavior of the `tree_ind` term.

[1,146] **rewrite** `reduce_set_ind` :
(let set(a, f).g = collect x: T. $a[x]$ in $b[a; f; g]$) ⟷
$b[T; (\lambda x.a[x]); (\lambda a2.$(let set($a$, f).g = $a[a2]$ in $b[a; f; g]))]$

15.2.4 Rules

Typehood and equality

The `set` term is a type in the Nuprl type theory. The `equal_set` and `isset_assum` rules define the `isset` well-formedness judgment. The `isset_assum` is added to the `trivialT` tactic for use as default reasoning.

[1,13] **rule** `set_type` {| intro_resource [] |} H:
[*ext*] $H\vdash$ set Type
[2,16] **rule** `equal_set` H:
[*ext*] $H\vdash$ s set ⟶
[*ext*] $H\vdash$ $s \in$ set
[1,5] **rule** `isset_assum` H J:
[*ext*] H; x: set; $J[x]\vdash$ x set

The `collect` terms are well-formed, if their index type T is a type in $\mathbb{U}_1$, and their element function a produces a set for any argument $x \in T$.

[4,57] **rule** `isset_collect` {| intro_resource [] |} H y:
[·] $H\vdash$ $T \in \mathbb{U}[1]$ ⟶
[·] H; y: $T\vdash$ $a[y]$ set ⟶
[*ext*] $H\vdash$ collect x: T. $a[x]$ set
[1,18] **rule** `isset_collect2` {| intro_resource [] |} H y:
[·] $H\vdash$ $T \in \mathbb{U}[1]$ ⟶
[·] H; y: $T\vdash$ $a[y]$ set ⟶
[*ext*] $H\vdash$ collect x: T. $a[x] \in$ set

Elimination

The elimination form performs induction on the assumption a: *set*. The inductive argument is this: goal C is true for any set a if it is true for some set $collect(x \in T.f(x))$ where the induction hypothesis is true on every child $f(x)$ for $x \in T$. By definition, induction requires that tree representing the set be well-founded (which is true for all W-types).

[9,139] **rule** `set_elim`
{| elim_resource [(ThinOption thinT)] |} H J a T f w z:

[*ext*]
1. *H*
2. a: set
3. *J*[*a*]
4. T: 𝕌[1]
5. f: *T* → set
6. w: ∀*x*:*T*. *C*[(*f x*)]
7. z: collect *x*: *T*. *f x* set
⊢ *C*[collect *x*: *T*. *f x*] ⟶
[*ext*] *H*; a: set; *J*[*a*]⊢ *C*[*a*]

Combinator equality

The induction combinator computes a value of type T if its argument z is a set, and the body $b[z, f, g]$ computes a value of type T, for any type $z \in \mathbb{U}_1$, any function $f \in z \to set$, and a recursive invocation $g \in x{:}\mathbb{U}_1 \to x \to T$.

[5,40]* **rule `set_ind_equality2` **{|** intro_resource [] **|}** H:
[wf**]** [·] *H*⊢ *z1* = *z2* ∈ set ⟶
[main**]** [·]
1. *H*
2. a1: 𝕌[1]
3. f1: *a1* → set
4. g1: x:𝕌[1] → *x* → *T*
⊢ *body1*[*a1*; *f1*; *g1*] = *body2*[*a1*; *f1*; *g1*] ∈ *T* ⟶

[*ext*]
1. *H*
⊢
(let set(*a1*, *f1*).*g1* = *z1* in *body1*[*a1*; *f1*; *g1*])
= (let set(*a2*, *f2*).*g2* = *z2* in *body2*[*a2*; *f2*; *g2*]) ∈ *T*

15.3 Czf_itt_eq module

The `Czf_itt_eq` module defines *extensional* equality of sets. Sets are equal if they have the same members, and in addition the equality must be *computable*. The basic definition of equality is given with the `eq` term, which is defined as follows.

$$eq(collect(x_1 \in T_1.f_1[x_1]), collect(x_2 \in T_2.f_2[x_2])) \equiv$$
$$\forall x_1{:}T_1.\exists x_2{:}T_2.eq(f_1[x_1], f_2[x_2])$$
$$\wedge \forall x_2{:}T_2.\exists x_1{:}T_2.eq(f_1[x_1], f_2[x_2])$$

This recursive definition requires that for each element of one set, there must be an equal element in the other set. The proof of an equality is a pair of functions that, given an element of one set, produce the equal element in the other set. Note that there is significant computational content in this judgment.

The `eq` judgment is a predicate in $\mathbb{U}_1$ for any two sets. The $\mathit{equal}(s_1, s_2)$ judgment adds the additional requirement $s_1 \in \mathit{set} \wedge s_2 \in \mathit{set}$ (which raises it to a predicate in $\mathbb{U}_2$).

In addition to the equality judgments, the `Czf_itt_eq` module also defines *functionality* judgments. The $\mathit{fun_set}(s.f[s])$ requires that the function f compute equal set values for equal set arguments. The $\mathit{fun_prop}(s.P[s])$ requires that for any two equal sets s_1 and s_2: $P[s_1] \Rightarrow P[s_2]$. The `dfun_prop` term is even more stringent. In the judgment $\mathit{dfun_prop}(A{:}\,A.B[x])$, the term A is a type, and $B[x]$ is a type for any $x \in A$. The judgment requires that a proof $B[u_2]$ be computable from *any* two proofs $u_1{:}\,A$ and $u_2{:}\,A$ and a proof $B[u_1]$:

$$\mathit{dfun_prop}(A{:}\,A.B[x]) \equiv u_1{:}\,A \to B[u_1] \to u_2{:}\,A \to B[u_2].$$

The `dfun_prop` term is used for functionality judgments on dependent types.

15.3.1 Parents

include Czf_itt_set

15.3.2 Terms

declare `eq` = Czf_itt_eq!eq{'s1; 's2}
declare `equal` = Czf_itt_eq!equal{'s1; 's2}
declare `fun_set` = Czf_itt_eq!fun_set{z. f['z]}
declare `fun_prop` = Czf_itt_eq!fun_prop{z. P['z]}
declare `dfun_prop`
    = Czf_itt_eq!dfun_prop{u. A['u]; x, y. B['x; 'y]}

15.3.3 Rewrites

The `eq` judgment requires that the two sets have the same elements.

! **rewrite** unfold_eq :
eq(*s1*; *s2*) ⟷
(let set(*T1*, *f1*).*g1* = *s1* in
let set(*T2*, *f2*).*g2* = *s2* in
(∀*y1*:*T1*. ∃*y2*:*T2*. eq((*f1 y1*); (*f2 y2*)))
∧ (∀*y2*:*T2*. ∃*y1*:*T1*. eq((*f1 y1*); (*f2 y2*))))

[1,240]* **rewrite reduce_eq :
eq((collect *x1*: *T1*. *f1*[*x1*]); (collect *x2*: *T2*. *f2*[*x2*])) ⟷
((∀*y1*:*T1*. ∃*y2*:*T2*. eq(*f1*[*y1*]; *f2*[*y2*]))
∧ (∀*y2*:*T2*. ∃*y1*:*T1*. eq(*f1*[*y1*]; *f2*[*y2*])))

The unfold_equal term requires that, in addition, the two arguments must be sets.

! **rewrite** unfold_equal :
(*s1* =' *s2*) ⟷
((*s1* set ∧ (*s2* set)) ∧ eq(*s1*; *s2*))

[1,224]* **rewrite reduce_equal :
(collect *x1*: *T1*. *f1*[*x1*] =' collect *x2*: *T2*. *f2*[*x2*]) ⟷
((collect *x1*: *T1*. *f1*[*x1*] set ∧ (collect *x2*: *T2*. *f2*[*x2*] set))
∧ (∀*y1*:*T1*. ∃*y2*:*T2*. eq(*f1*[*y1*]; *f2*[*y2*]))
∧ (∀*y2*:*T2*. ∃*y1*:*T1*. eq(*f1*[*y1*]; *f2*[*y2*])))

The following four rewrites define the functionality judgments.

! **rewrite** unfold_fun_set :
(∀*z*.*f*[*z*] fun_set) ⟷
(∀*s1*:set. ∀*s2*:set. ((*s1* =' *s2*) → (*f*[*s1*] =' *f*[*s2*])))

! **rewrite** unfold_fun_prop :
(∀*z*.*P*[*z*] fun_prop) ⟷
(∀*s1*:set. ∀*s2*:set. ((*s1* =' *s2*) → *P*[*s1*] → *P*[*s2*]))

! **rewrite** unfold_dfun_prop2 :
(∀*x*:*A*. *B*[*x*] fun_prop) ⟷
(u1:*A* → *B*[*u1*] → u2:*A* → *B*[*u2*])

! **rewrite** unfold_dfun_prop :
(∀*u*:*A*[*u*]. *x*,*y*.*B*[*x*; *y*] fun_prop) ⟷
(∀*s1*:set
∀*s2*:set
((*s1* =' *s2*) → (u1:*A*[*s1*] → *B*[*s1*; *u1*] → u2:*A*[*s2*] → *B*[*s2*; *u2*])))

15.3.4 Rules

Typehood and equality

The `eq` judgment is well-formed if both arguments are sets. The *equality* if the `eq` judgment requires the set arguments to be equal (in the native `set` type).

[10,355]* **rule `eq_equality1` {| intro_resource [] |} H:
[·] $H\vdash$ *s1* set $\longrightarrow$
[·] $H\vdash$ *s2* set $\longrightarrow$
[*ext*] $H\vdash$ eq(*s1*; *s2*) $\in \mathbb{U}[1]$
[1,14]* **rule `eq_type` {| intro_resource [] |} H:
[·] $H\vdash$ *s1* set $\longrightarrow$
[·] $H\vdash$ *s2* set $\longrightarrow$
[*ext*] $H\vdash$ eq(*s1*; *s2*) Type
[12,180]* **rule `eq_equality2` {| intro_resource [] |} H:
[·] $H\vdash$ *s1* = *s3* $\in$ set $\longrightarrow$
[·] $H\vdash$ *s2* = *s4* $\in$ set $\longrightarrow$
[*ext*] $H\vdash$ eq(*s1*; *s2*) = eq(*s3*; *s4*) $\in \mathbb{U}[1]$

The `eq` judgment also requires the arguments to be sets, but in addition, and equality judgment *equal*(s_1, s_2) *implies* that both s_1 and s_2 are types.

[1,77]* **rule `equal_type` {| intro_resource [] |} H:
[·] $H\vdash$ *s1* set $\longrightarrow$
[·] $H\vdash$ *s2* set $\longrightarrow$
[*ext*] $H\vdash$ (*s1* =' *s2*) Type
[3,45]* **rule `equal_intro` {| intro_resource [] |} H:
[wf**]** [·] $H\vdash$ *s1* set $\longrightarrow$
[wf**]** [·] $H\vdash$ *s2* set $\longrightarrow$
[*ext*] $H\vdash$ eq(*s1*; *s2*) $\longrightarrow$
[*ext*] $H\vdash$ *s1* =' *s2*
[2,22]* **rule `equal_isset_left` H *s2*:
[*ext*] $H\vdash$ *s1* =' *s2* $\longrightarrow$
[*ext*] $H\vdash$ *s1* set
[2,22]* **rule `equal_isset_right` H *s1*:
[*ext*] $H\vdash$ *s1* =' *s2* $\longrightarrow$
[*ext*] $H\vdash$ *s2* set

Equality is an equivalence relation

The `eq` judgment is reflexive, symmetric, and transitive.

[7,262]* **rule eq_ref {| intro_resource [] |} H:
[·] $H\vdash$ *s1* set ⟶
[*ext*] $H\vdash$ eq(*s1*; *s1*)
[16,618]* **rule eq_sym H:
[wf**]** [·] $H\vdash$ *s1* set ⟶
[wf**]** [·] $H\vdash$ *s2* set ⟶
[*ext*] $H\vdash$ eq(*s2*; *s1*) ⟶
[*ext*] $H\vdash$ eq(*s1*; *s2*)
[22,980]* **rule eq_trans H *s2*:
[wf**]** [·] $H\vdash$ *s1* set ⟶
[wf**]** [·] $H\vdash$ *s2* set ⟶
[wf**]** [·] $H\vdash$ *s3* set ⟶
[*ext*] $H\vdash$ eq(*s1*; *s2*) ⟶
[*ext*] $H\vdash$ eq(*s2*; *s3*) ⟶
[*ext*] $H\vdash$ eq(*s1*; *s3*)

Functionality

The *fun_set*($z.f[z]$) judgment implies that $f[z]$ is a set for any set z.

[6,105]* **rule eq_isset H J ($\forall z.f[z]$ fun_set):
[*ext*] *H*; z: set; *J*[z]$\vdash$ $\forall z.f[z]$ fun_set ⟶
[*ext*] *H*; z: set; *J*[z]$\vdash$ $f[z]$ set

The eq and equal judgments are *functional* with respect to their set arguments.

[29,789]* **rule eq_fun {| intro_resource [] |} H:
[*ext*] $H\vdash$ $\forall z.f1[z]$ fun_set ⟶
[*ext*] $H\vdash$ $\forall z.f2[z]$ fun_set ⟶
[*ext*] $H\vdash$ $\forall z.$eq($f1[z]$; $f2[z]$) fun_prop
[29,732]* **rule equal_fun {| intro_resource [] |} H:
[*ext*] $H\vdash$ $\forall z.f1[z]$ fun_set ⟶
[*ext*] $H\vdash$ $\forall z.f2[z]$ fun_set ⟶
[*ext*] $H\vdash$ $\forall z.(f1[z] =' f2[z])$ fun_prop

Substitution

The following two rules define substitution. Set s_1 can be replaced by set s_2 in a context $P[s_1]$ if s_1 an s_2 are equal, and the context $P[x]$ is *functional* on set arguments.

[7,159]* **rule eq_hyp_subst H J *s1 s2* bind($v.P[v]$) z:
[*ext*] *H*; x: *P*[*s1*]; *J*[x]$\vdash$ *s1* =' *s2* ⟶

[*ext*] *H*; x: *P*[*s1*]; *J*[*x*]; z: *P*[*s2*]⊢ *C*[*x*] ⟶
[*ext*] *H*; x: *P*[*s1*]; *J*[*x*]⊢ ∀*z*.*P*[*z*] fun_prop ⟶
[*ext*] *H*; x: *P*[*s1*]; *J*[*x*]⊢ *C*[*x*]
[7,197]* **rule `eq_concl_subst` H *s1 s2* bind(*v*.*C*[*v*]) z:
[*ext*] *H*⊢ *s1* =' *s2* ⟶
[*ext*] *H*⊢ *C*[*s2*] ⟶
[*ext*] *H*⊢ ∀*z*.*C*[*z*] fun_prop ⟶
[*ext*] *H*⊢ *C*[*s1*]

Typehood of the functionality judgments

The `fun_set` judgment requires that it's argument be a family of sets, and the `fun_prop` judgment requires that it's argument be a family of propositions.

[2,103]* **rule `fun_set_type` {| intro_resource [] |} H:
[·] *H*; z: set⊢ *f*[*z*] set ⟶
[*ext*] *H*⊢ ∀*z*.*f*[*z*] fun_set Type
[1,135]* **rule `fun_prop_type` {| intro_resource [] |} H:
[·] *H*; z: set⊢ *f*[*z*] Type ⟶
[*ext*] *H*⊢ ∀*z*.*f*[*z*] fun_prop Type

The trivial cases, where the functionality argument does not depend on the set argument, are functional. The identity function is also functional.

[1,87]* **rule `fun_set` {| intro_resource [] |} H:
[·] *H*⊢ *u* set ⟶
[*ext*] *H*⊢ ∀*z*.*u* fun_set
[1,34]* **rule `fun_ref` {| intro_resource [] |} H:
[*ext*] *H*⊢ ∀*z*.*z* fun_set
[1,54]* **rule `fun_prop` {| intro_resource [] |} H:
[·] *H*⊢ *P* Type ⟶
[*ext*] *H*⊢ ∀*z*.*P* fun_prop

Substitution

`setSubstT` The `setSubstT` tactic *substitutes* one set for another. The usage is `setSubstT` i $eq(s_1, s_2)$, which replaces all occurrences of the term s_1 with the term s_2 in clause i.

15.3.5 Tactics

`eqSetRefT`, `eqSetSymT`, `eqSetTransT` The three `eqSet` tactics apply equivalence relation reasoning for the `eq` set judgment.

15.4 Czf_itt_member module

The `Czf_itt_member` module defines membership in a set. The basic definition is an existential judgment: a set s is an element of a set $collect(x \in T.f[x])$ if there is some element $a: T$ and $eq(s, f[a])$.

Note that equality has to be defined *before* membership. We also prove the *extensionality* judgment here; two sets are equal if they have the same members.

15.4.1 Parents

include Czf_itt_eq

15.4.2 Terms

The `member` term defines the membership judgment.

declare `mem` = Czf_itt_member!mem{'x; 'y}
declare `member` = Czf_itt_member!member{'x; 'y}

15.4.3 Rewrites

The `member` judgment is defined using the `set_ind` induction combinator.

! **rewrite** `unfold_mem` :
$(x \in\text{s } y) \longleftrightarrow$
$(\text{let set}(T, f).g = y \text{ in } \exists t{:}T.\ \text{eq}(x; (f\ t)))$

[1,74]* **rewrite `reduce_mem` :
$(x \in\text{s collect } y{:}\ T.\ f[y]) \longleftrightarrow$
$(\exists t{:}T.\ \text{eq}(x; f[t]))$

15.4.4 Rules

Well-formedness

The `member` judgment is well-formed if-and-only-if its arguments are sets.

[6,129]* **rule mem_type {| intro_resource [] |} H:
  [wf**]** [·] $H \vdash$ *s1* set $\longrightarrow$
  [wf**]** [·] $H \vdash$ *s2* set $\longrightarrow$
  [*ext*] $H \vdash$ *s1* $\in$s *s2* Type
[7,156]* **rule mem_equal {| intro_resource [] |} H:
  [wf**]** [·] $H \vdash$ *s1* set $\longrightarrow$
  [wf**]** [·] $H \vdash$ *s2* set $\longrightarrow$
  [*ext*] $H \vdash$ *s1* $\in$s *s2* $\in \mathbb{U}$[1]
[3,45]* **rule member_intro {| intro_resource [] |} H:
  [wf**]** [·] $H \vdash$ *s1* set $\longrightarrow$
  [wf**]** [·] $H \vdash$ *s2* set $\longrightarrow$
  [*ext*] $H \vdash$ *s1* $\in$s *s2* $\longrightarrow$
  [*ext*] $H \vdash$ *s1* $\in$S *s2*
[2,24]* **rule elem_isset H *y*:
  [*ext*] $H \vdash x \in$S $y \longrightarrow$
  [*ext*] $H \vdash x$ set
[2,24]* **rule set_isset H *x*:
  [*ext*] $H \vdash x \in$S $y \longrightarrow$
  [*ext*] $H \vdash y$ set

Functionality

The member judgment is functional in both its arguments. The next two rules provide simple functionality judgments for the two set arguments.

[10,397]* **rule mem_fun_left H *s1*:
  [wf**]** [·] $H \vdash$ *s1* set $\longrightarrow$
  [wf**]** [·] $H \vdash$ *s2* set $\longrightarrow$
  [wf**]** [·] $H \vdash$ *s3* set $\longrightarrow$
  [*ext*] $H \vdash$ eq(*s1*; *s2*) $\longrightarrow$
  [*ext*] $H \vdash$ *s1* $\in$s *s3* $\longrightarrow$
  [*ext*] $H \vdash$ *s2* $\in$s *s3*

The member_fun rule proves the general functionality judgment.

[20,455]* **rule member_fun {| intro_resource [] |} H:
  [*ext*] $H \vdash \forall z.f1[z]$ fun_set $\longrightarrow$
  [*ext*] $H \vdash \forall z.f2[z]$ fun_set $\longrightarrow$
  [*ext*] $H \vdash \forall z.f1[z] \in$s $f2[z]$ fun_prop

Set extensionality

Two sets are equal if-and-only-if they have the same elements. The proof of this theorem is straightforward. The two membership goals

are the functions that "choose," for any element of one set, an equal element in the other set. The equality judgment can be proved with the pair of both choice functions.

[20,998]* **rule `set_ext` H x y:
[wf**]** [*ext*] $H \vdash$ *s1* set $\longrightarrow$
[wf**]** [*ext*] $H \vdash$ *s2* set $\longrightarrow$
[main**]** [*ext*] H; x: set; y: $x \in_s$ *s1* $\vdash x \in_s$ *s2* $\longrightarrow$
[main**]** [*ext*] H; x: set; y: $x \in_s$ *s2* $\vdash x \in_s$ *s1* $\longrightarrow$
[*ext*] $H \vdash$ eq(*s1*; *s2*)

15.4.5 Tactics

`memberOfT`, `setOfT` The `memberOfT` applies the `elem_isset` rule, and the `setOfT` tactic applies the `set_isset` rule. Both tactics infer the well-formedness of a set for a membership or equality judgment.

`setExtT` The `setExtT` tactic refines a set-equality goal using the rule of *extensionality* `set_ext`. This rule is not added to the `dT` tactic for default reasoning because in many cases, equality judgments can be proved in simpler ways.

15.5 Czf_itt_fol module

The `Czf_itt_fol` module defines the first-order logic for the set theory. The predicates are defined as follows.

$$\begin{array}{rcl} false_s & \equiv & Void \\ true_s & \equiv & Unit \\ A \vee_s B & \equiv & A \cup B \\ A \wedge_s B & \equiv & A \times B \\ A \Rightarrow_s B & \equiv & A \rightarrow B \\ \exists_s x{:}A.B[x] & \equiv & x{:}A \times B[x] \\ \forall_s x{:}A.B[x] & \equiv & x{:}A \rightarrow B[x] \end{array}$$

The first-order logic is defined in seven modules, one for each logical operator: `Czf_itt_false`, `Czf_itt_true`, `Czf_itt_or`, `Czf_itt_and`, `Czf_itt_implies`, `Czf_itt_exists`, and `Czf_itt_all`.

We omit the modules, the proofs are straightforward. The goal of these modules is to prove *functionality* and *restriction* of each of the operators. The restriction judgment follows directly from the well-formedness of the logical operators. The disjunction demonstrates a typical argument of functionality for a propositional operator.

Functionality of disjunction The goal is $\mathit{fun_prop}(z.A[z] \vee_s B[z])$, assuming $\mathit{fun_prop}(z.A[z])$ and $\mathit{fun_prop}(z.B[z])$. To prove this, assume $A[s_1] \vee_s B[s_1]$ for some set s_1, and prove $A[s_2] \vee_s B[s_2]$ for some set $s_2 = s_1$.

First, perform a case analysis on $A[s_1] \vee_s B[s_1]$. In the case where $A[s_1]$, a proof of $A[s_2]$ can be constructed from the proof of functionality of A. This proof is sufficient for proving $A[s_2] \vee_s B[s_2]$. The case for B is similar.

The quantifiers are somewhat more difficult. Strictly speaking, the quantifiers are unnecessary, because the set theory can't reason about the programs in the first-order logic directly. The usual set-theoretic quantifiers are defined in the `Czf_itt_dall`, `Czf_itt_dexists`, `Czf_itt_sall`, and `Czf_itt_sexists` modules. The existential demonstrates a typical functionality argument.

Functionality of existential quantification The goal is to prove $\mathit{fun_prop}(z.w{:}A[z] \times B[z, w])$, given $\mathit{fun_prop}(z.A[z])$ and $\mathit{dfun_prop}(A[z]{:}A[z].B[z, w])$ for any set z. This means that we must produce a proof of $w{:}A[z_1] \times B[z_1, w]$ from a proof of $w{:}A[z_2] \times B[z_2, w]$ for any two equal sets z_1 and z_2.

The assumption $\mathit{fun_prop}(z.A[z])$ means that a proof of $A[z_2]$ can be computed from a proof $A[z_1]$ for any equal sets z_1 and z_2. The second assumption means that a proof $B[z_2, w_2]$ can be found given *any* proofs $w_1{:}A[z_1]$, $w_2{:}A[z_2]$, and $B[z_1; w_1]$. The proof of the goal is constructed from the proof $w_2{:}A[z_2]$ and the proof of $B[z_2, w_2]$.

15.6 Czf_itt_dall module

The `Czf_itt_dall` theory defines *restricted* universal quantification. The syntax of the operator is $\forall x \in_s s.P[x]$, where s is a set, and $P[x]$ is a functional proposition for any set argument x. The proposition is true if $P[a]$ is true for any element $a \in_s s$.

The restricted universal quantification is coded as an implication on the elements of s.

$$\forall x \in_s \mathit{collect}(y \in T.f[y]).P[x] \equiv \forall x{:}T.P[f(x)]$$

15.6.1 Parents

include Czf_itt_all
include Czf_itt_set_ind

15.6.2 Terms

declare `dall` = Czf_itt_dall!dall{'T; x. A['x]}

15.6.3 Rewrites

The `dall` term is defined by set induction on the set argument as a universal quantification over the index type.

! **rewrite** `unfold_dall` :
$(\forall x \in \mathrm{s}\ s.A[x]) \longleftrightarrow$
$(\mathrm{let\ set}(a, f).g = s\ \mathrm{in}\ (\mathrm{x}{:}a \to A[(f\ x)]))$

[1,64]* **rewrite `reduce_dall` :
$(\forall y \in \mathrm{s}\ \mathrm{collect}\ x{:}\ T.\ f[x].A[y]) \longleftrightarrow$
$(\mathrm{t}{:}T \to A[f[t]])$

15.6.4 Rules

Well-formedness

The $\forall x \in_s s.P[x]$ proposition is well-formed if s is a set, and $P[x]$ is a proposition for any set x.

[5,155]* **rule `dall_type` **{|** intro_resource [] **|}** H y:
[wf**]** [·] $H \vdash s$ set $\longrightarrow$
[wf**]** [·] H; y: set$\vdash A[y]$ Type $\longrightarrow$
[*ext*] $H \vdash (\forall x \in \mathrm{s}\ s.A[x])$ Type

Introduction

The proposition $\forall x \in_s s.P[x]$ is true if it is well-formed, and $P[a]$ is true for any set $a \in_s s$.

[14,628]* **rule `dall_intro` **{|** intro_resource [] **|}** H a b:
[wf**]** [·] $H \vdash s$ set $\longrightarrow$
[wf**]** [·] H; a: set$\vdash A[a]$ Type $\longrightarrow$
[main**]** [*ext*] H; a: set; b: $a \in \mathrm{s}\ s \vdash A[a] \longrightarrow$
[*ext*] $H \vdash \forall x \in \mathrm{s}\ s.A[x]$

Elimination

The elimination form instantiates the universal quantification $\forall x \in_s s.P[x]$ on a particular argument $z \in_s s$. It produces an additional assumption $A[z]$.

[21,629]* **rule `dall_elim` **{|** elim_resource [] **|}** H J z w:
[wf**]** [$\cdot$] H; x: $\forall y \in$s $s.A[y]$; $J[x]$; w: set$\vdash$ $A[w]$ Type $\longrightarrow$
[wf**]** [*ext*] H; x: $\forall y \in$s $s.A[y]$; $J[x]\vdash$ $\forall w.A[w]$ fun_prop $\longrightarrow$
[antecedent**]** [*ext*] H; x: $\forall y \in$s $s.A[y]$; $J[x]\vdash$ $z \in$S s $\longrightarrow$
[main**]** [*ext*] H; x: $\forall y \in$s $s.A[y]$; $J[x]$; w: $A[z]\vdash$ $C[x]$ $\longrightarrow$
[*ext*] H; x: $\forall y \in$s $s.A[y]$; $J[x]\vdash$ $C[x]$

Functionality

The `dall` proposition is functional in it's set and proposition arguments.

[27,1453]* **rule `dall_fun` **{|** intro_resource [] **|}** H:
[*ext*] $H\vdash$ $\forall z.A[z]$ fun_set $\longrightarrow$
[*ext*] H; z: set$\vdash$ $\forall x.B[z; x]$ fun_prop $\longrightarrow$
[*ext*] H; z: set$\vdash$ $\forall x.B[x; z]$ fun_prop $\longrightarrow$
[wf**]** [$\cdot$] H; z: set; x: set$\vdash$ $B[z; x]$ Type $\longrightarrow$
[*ext*] $H\vdash$ $\forall z.(\forall y \in$s $A[z].B[z; y])$ fun_prop

Restriction

The proposition $\forall x \in_s s.P[x]$ is restricted for any set s because the proposition quantifies over the *index* type of the set argument (which is a restricted proposition itself).

[7,96]* **rule `dall_res` **{|** intro_resource [] **|}** H:
[wf**]** [$\cdot$] $H\vdash$ A set $\longrightarrow$
[$\cdot$] H; x: set$\vdash$ <$B[x]$> $\longrightarrow$
[*ext*] $H\vdash$ <$(\forall y \in$s $A.B[y])$>

15.7 Czf_itt_dexists module

The `Czf_itt_dexists` theory defines *restricted* existential quantification. The syntax of the operator is $\exists x \in_s s.P[x]$, where s is a set, and $P[x]$ is a functional proposition for any set argument x. The proposition is true if $P[a]$ is true for *some* element $a \in_s s$.

The restricted universal quantification is coded as an implication on the elements of s.

$$\exists x \in_s \mathit{collect}(y \in T.f[y]).P[x] \equiv x{:}T \times P[x]$$

15.7.1 Parents

include Czf_itt_sep
include Czf_itt_set_ind

15.7.2 Terms

declare `dexists` = Czf_itt_dexists!dexists{'T; x. A['x]}

15.7.3 Rewrites

The existential is defined by set induction on the set argument as a dependent product type.

! **rewrite** `unfold_dexists` :
(∃x ∈s s.$A[x]$) ⟷
(let set(T, f).g = s in (x:T × $A[(f\ x)]$))

[1,64]* **rewrite `reduce_dexists` :
(∃y ∈s collect x: T. $f[x]$.$A[y]$) ⟷
(t:T × $A[f[t]]$)

15.7.4 Rules

Well-formedness

The proposition ∃x ∈$_s$ s.$P[x]$ is well-formed if s is a set, and $P[x]$ is a well-formed proposition for *any* set argument x.

[5,155]* **rule `dexists_type` **{|** intro_resource [] **|}** H y:
[wf**]** [·] H⊢ s set ⟶
[wf**]** [·] H; y: set⊢ $A[y]$ Type ⟶
[*ext*] H⊢ (∃x ∈s s.$A[x]$) Type

Introduction

The existential $\exists x \in_s s.P[x]$ is true if it is well-formed and if $P[a]$ is true for some element $a \in_s s$.

[13,738]* **rule `dexists_intro` {| intro_resource [] |} H z w:
[wf**]** [$\cdot$] H; w: set$\vdash$ $A[w]$ Type $\longrightarrow$
[wf**]** [*ext*] $H\vdash$ $\forall x.A[x]$ fun_prop $\longrightarrow$
[main**]** [*ext*] $H\vdash$ $z \in$S s $\longrightarrow$
[antecedent**]** [*ext*] $H\vdash$ $A[z]$ $\longrightarrow$
[*ext*] $H\vdash$ $\exists x \in$s $s.A[x]$

Elimination

The proof of the existential $\exists x \in_s s.P[x]$ has two parts: an element $a \in_s s$, and a proof $P[a]$. The elimination form produces these parts.

[11,497]* **rule `dexists_elim` {| elim_resource [] |} H J x z v w:
[wf**]** [$\cdot$] H; x: $\exists y \in$s $s.A[y]$; $J[x]\vdash$ s set $\longrightarrow$
[wf**]** [$\cdot$] H; x: $\exists y \in$s $s.A[y]$; $J[x]$; z: set$\vdash$ $A[z]$ Type $\longrightarrow$
[wf**]** [*ext*] H; x: $\exists y \in$s $s.A[y]$; $J[x]\vdash$ $\forall z.A[z]$ fun_prop $\longrightarrow$
[main**]** [*ext*]
1. H
2. x: $\exists y \in$s $s.A[y]$
3. $J[x]$
4. z: set
5. v: $z \in$s s
6. w: $A[z]$

$\vdash$ $C[x]$ $\longrightarrow$
[*ext*] H; x: $\exists y \in$s $s.A[y]$; $J[x]\vdash$ $C[x]$

Functionality

The existential is functional in both its set and proposition arguments.

[13,771]* **rule `dexists_fun` {| intro_resource [] |} H:
[*ext*] $H\vdash$ $\forall z.A[z]$ fun_set $\longrightarrow$
[*ext*] H; z: set$\vdash$ $\forall x.B[z; x]$ fun_prop $\longrightarrow$
[*ext*] H; z: set$\vdash$ $\forall x.B[x; z]$ fun_prop $\longrightarrow$
[wf**]** [$\cdot$] H; z: set; x: set$\vdash$ $B[z; x]$ Type $\longrightarrow$
[*ext*] $H\vdash$ $\forall z.(\exists y \in$s $A[z].B[z; y])$ fun_prop

Restriction

The existential is a restricted formula because it is a quantification over the *index* type of the set argument.

[5,94]* **rule `dexists_res2` **{|** intro_resource [] **|}** H:
[wf**]** [·] $H \vdash A$ set $\longrightarrow$
[·] H; u: set$\vdash$ <$B[u]$> $\longrightarrow$
[*ext*] $H \vdash$ <($\exists y \in \mathrm{s}\ A.B[y]$)>

15.8 Czf_itt_sall module

The `Czf_itt_sall` module defines the *unrestricted* universal quantification $\forall_s x.P[x]$ over all sets x. The proposition $P[x]$ must be well-formed for all sets x.

15.8.1 Parents

include Czf_itt_set

15.8.2 Terms

declare `sall` = Czf_itt_sall!sall{x. A['x]}

15.8.3 Rewrites

The quantification $\forall_s x.P[x]$ is defined using the universal quantifier `all` from the `Itt_logic` module.

! **rewrite** `unfold_sall` : $(\forall_s x.A[x]) \longleftrightarrow (\forall x{:}\mathrm{set}.\ A[x])$

15.8.4 Rules

Well-formedness

The quantification $\forall_s x.A[x]$ is well-formed if $A[x]$ is a proposition for any set x.

[1,61]* **rule `sall_type` **{|** intro_resource [] **|}** H y:
[*ext*] H; y: set$\vdash$ $A[y]$ Type $\longrightarrow$
[*ext*] $H \vdash (\forall_s x.A[x])$ Type

Introduction

The quantification $\forall_s x.A[x]$ is true if it is well-formed, and if $A[x]$ is true for every set x.

*[1,13] **rule** sall_intro {| intro_resource [] |} H a:
[*ext*] H; a: set$\vdash$ $A[a]$ $\longrightarrow$
[*ext*] $H\vdash$ $\forall_s x.A[x]$

Elimination

The elimination form instantiates the universal assumption on a particular set argument z.

*[3,41] **rule** sall_elim {| elim_resource [] |} H J x z w:
[wf**]** [$\cdot$] H; x: $\forall_s y.A[y]$; $J[x]\vdash$ z set $\longrightarrow$
[main**]** [*ext*] H; x: $\forall_s y.A[y]$; $J[x]$; w: $A[z]\vdash$ $T[x]$ $\longrightarrow$
[*ext*] H; x: $\forall_s y.A[y]$; $J[x]\vdash$ $T[x]$

15.9 Czf_itt_sexists module

The Czf_itt_sexists module defines the *unrestricted* existential quantification $\exists_s x.P[x]$. The proposition $P[x]$ must be well-formed for any set argument. The existential is true, if $P[a]$ is true for some set a.

15.9.1 Parents

include Czf_itt_and

15.9.2 Terms

declare sexists = Czf_itt_sexists!sexists{x. A['x]}

15.9.3 Rewrites

The unrestricted existential is defined with the type-theoretic existential exists from the Itt_logic module.

! **rewrite** unfold_sexists : $(\exists_s x.A[x]) \longleftrightarrow (\exists x$:set. $A[x])$

15.9.4 Rules

Well-formedness

The unrestricted existential $\exists_s x.P[x]$ is well-formed if $P[x]$ is a well-formed proposition for any set argument x.

[1,13]* **rule `sexists_type` **{|** intro_resource [] **|}** H y:
[·] H; y: set$\vdash$ $A[y]$ Type $\longrightarrow$
[*ext*] $H\vdash$ ($\exists_s x.A[x]$) Type

Introduction

The existential $\exists_s x.P[x]$ is true if $P[a]$ is true for some set a.

[3,90]* **rule `sexists_intro` **{|** intro_resource [] **|}** H z w:
[wf**]** [*ext*] $H\vdash$ z set $\longrightarrow$
[main**]** [*ext*] $H\vdash$ $A[z]$ $\longrightarrow$
[wf**]** [*ext*] H; w: set$\vdash$ $A[w]$ Type $\longrightarrow$
[*ext*] $H\vdash$ $\exists_s x.A[x]$

Elimination

The proof of the existential $\exists_s x.P[x]$ is a pair of a witness set a and a proof $P[a]$.

[1,8]* **rule `sexists_elim` **{|** elim_resource [] **|}** H J x z w:
[*ext*] H; z: set; w: $A[z]$; $J[(z,w)]\vdash$ $T[(z,w)]$ $\longrightarrow$
[*ext*] H; x: $\exists_s y.A[y]$; $J[x]\vdash$ $T[x]$

15.10 Czf_itt_subset module

The `Czf_itt_subset` module defines the subset proposition $s_1 \subseteq_s s_2$, which is a proposition for any two sets s_1 and s_2. The subset is a derived proposition defined as follows.

$$s_1 \subseteq_s s_2 \equiv \forall x \in_s s_1.x \in_s s_2$$

15.10.1 Parents

include Czf_itt_dall

15.10.2 Terms

declare `subset` = Czf_itt_subset!subset{'s1; 's2}

15.10.3 Rewrites

The subset is defined using restricted universal quantification.

! **rewrite** `unfold_subset` : $(s1 \subseteq_s s2) \longleftrightarrow (\forall x \in_s s1. x \in_s s2)$

15.10.4 Rules

Well-formedness

The subset proposition $s_1 \subseteq_s s_2$ is well-formed if s_1 and s_2 are both sets.

[1,28]* **rule `subset_type` **{|** intro_resource [] **|}** H:
[wf**]** [·] $H \vdash$ *s1* set $\longrightarrow$
[wf**]** [·] $H \vdash$ *s2* set $\longrightarrow$
[*ext*] $H \vdash$ $(s1 \subseteq_s s2)$ Type

Introduction

The subset proposition $s_1 \subseteq_s s_2$ is true if every element $x \in_s s_1$ is also an element of s_2.

[1,31]* **rule `subset_intro` **{|** intro_resource [] **|}** H x:
[wf**]** [·] $H \vdash$ *s1* set $\longrightarrow$
[wf**]** [·] $H \vdash$ *s2* set $\longrightarrow$
[main**]** [*ext*] H; x: set; y: $x \in_s s1 \vdash x \in_s s2 \longrightarrow$
[*ext*] $H \vdash s1 \subseteq_s s2$

Elimination

The elimination form of the proposition $s_1 \subseteq_s s_2$ takes an element $x \in_s s_1$ and it produces a proof that $x \in_s s_2$.

[3,154]* **rule `subset_elim` **{|** elim_resource [] **|}** H J *s* z:
[wf**]** [·] H; x: $s1 \subseteq_s s2$; $J[x] \vdash s$ set $\longrightarrow$
[wf**]** [·] H; x: $s1 \subseteq_s s2$; $J[x] \vdash s1$ set $\longrightarrow$
[wf**]** [·] H; x: $s1 \subseteq_s s2$; $J[x] \vdash s2$ set $\longrightarrow$
[antecedent**]** [*ext*] H; x: $s1 \subseteq_s s2$; $J[x] \vdash s \in_s s1 \longrightarrow$
[main**]** [*ext*] H; x: $s1 \subseteq_s s2$; $J[x]$; z: $s \in_s s2 \vdash C[x] \longrightarrow$
[*ext*] H; x: $s1 \subseteq_s s2$; $J[x] \vdash C[x]$

Functionality

The subset proposition is functional in both set arguments, and it is a restricted proposition.

[1,28]* **rule `subset_res` **{|** intro_resource [] **|}** H:
[wf**]** [·] $H \vdash$ *s1* set $\longrightarrow$
[wf**]** [·] $H \vdash$ *s2* set $\longrightarrow$
[*ext*] $H \vdash$ <(*s1* $\subseteq$s *s2*)>
[3,89]* **rule `subset_fun` **{|** intro_resource [] **|}** H:
[*ext*] $H \vdash \forall z.s1[z]$ fun_set $\longrightarrow$
[*ext*] $H \vdash \forall z.s2[z]$ fun_set $\longrightarrow$
[*ext*] $H \vdash \forall z.(s1[z] \subseteq\text{s } s2[z])$ fun_prop

15.11 Czf_itt_sep module

The `Czf_itt_sep` module defines *restricted separation*. Separation is defined as a set constructor $\{x \in_s s \mid P[x]\}$. The term s must be a set, and $P[x]$ must be a functional, restricted proposition. The elements of the separation are the elements of $x \in s$ for which $P[x]$ is true.

The separation constructor is a set if s is a set, and if $P[x]$ is *restricted*. Our formalization of restriction differs slightly from Aczel's account, although both are equivalent. In Aczel's definition, a proposition $P[x]$ is restricted if it contains no *unbounded* quantifiers $\forall s \ldots$ and $\exists s \ldots$. In our construction, a predicate P is restricted if it is a well-formed proposition in $\mathbb{U}_1$.

The separation constructor $\{z \in_s \mathit{collect}(x \in T.f[x]) \mid P[z]\}$ is defined with the product type as the set $\mathit{collect}(z \in (x{:}A \times P[x]).f(\mathit{fst}(z)))$.

15.11.1 Parents

include Czf_itt_member

15.11.2 Terms

declare `sep` = Czf_itt_sep!sep{'s; x. P['x]}
declare `restricted` = Czf_itt_sep!restricted{'P}

15.11.3 Rewrites

The `sep` term is defined by set induction.

! **rewrite** `unfold_sep` :
{ $x \in\mathrm{s}\ s \mid P[x]$ } $\longleftrightarrow$
(let set(T, f).g = s in collect z: (t:$T \times P[(f\ t)]$). $f\ z$.1)

[1,100]* **rewrite `reduce_sep` :
{ $z \in\mathrm{s}$ (collect x: T. $f[x]$) | $P[z]$ } $\longleftrightarrow$
(collect w: (t:$T \times P[f[t]]$). $f[w.1]$)

The *restricted*(P) predicate means that the proposition is well-formed in $\mathbb{U}_1$.

! **rewrite** `unfold_restricted` : (<P>) $\longleftrightarrow$ ($P \in \mathbb{U}[1]$)

Equality and membership are restricted judgments

The next two rules show that equality and membership are restricted for any `set` arguments.

[8,239]* **rule `eq_restricted` **{|** intro_resource [] **|}** H:
[wf**]** [·] $H\vdash$ *s1* set $\longrightarrow$
[wf**]** [·] $H\vdash$ *s2* set $\longrightarrow$
[*ext*] $H\vdash$ <eq(*s1*; *s2*)>
[1,13]* **rule `member_restricted` **{|** intro_resource [] **|}** H:
[wf**]** [·] $H\vdash$ *s1* set $\longrightarrow$
[wf**]** [·] $H\vdash$ *s2* set $\longrightarrow$
[*ext*] $H\vdash$ <*s1* $\in$s *s2*>

Well-formedness

The separation $\{x \in_s s \mid P[x]\}$ is well-formed if s is a set, and $P[x]$ is restricted and functional on any set argument x.

[7,217]* **rule `sep_isset` **{|** intro_resource [] **|}** H z:
[wf**]** [·] $H\vdash$ s set $\longrightarrow$
[wf**]** [*ext*] $H\vdash$ $\forall z.P[z]$ fun_prop $\longrightarrow$
[wf**]** [·] H; z: set$\vdash$ <$P[z]$> $\longrightarrow$
[*ext*] $H\vdash$ { $x \in\mathrm{s}\ s \mid P[x]$ } set

Introduction

A set x is a member of $\{z \in_s s \mid P[z]\}$ if the separation is well-formed; if $x \in_S s$; and $P[x]$.

[19,710]* **rule `sep_intro2` **{|** intro_resource [] **|}** H:
[wf**]** [·] H; w: set$\vdash$ <$P[w]$> $\longrightarrow$
[wf**]** [*ext*] $H\vdash$ $\forall z.P[z]$ fun_prop $\longrightarrow$
[main**]** [*ext*] $H\vdash$ $x \in$S s $\longrightarrow$
[main**]** [*ext*] $H\vdash$ $P[x]$ $\longrightarrow$
[*ext*] $H\vdash$ $x \in$s { $z \in$s s | $P[z]$ }

Elimination

An assumption $x \in_s \{y \in_s s \mid P[y]\}$ implies two facts: $x \in_s s$ and $P[x]$. The computational content of the predicate $P[x]$ is visible (unlike the separation "set" constructor in the Nuprl type theory module `Itt_set`).

[21,648]* **rule `sep_elim` **{|** elim_resource [] **|}** H J u v z:
[wf**]** [·] H; w: $x \in$s { $y \in$s s | $P[y]$ }; $J[w]\vdash$ x set $\longrightarrow$
[wf**]** [·] H; w: $x \in$s { $y \in$s s | $P[y]$ }; $J[w]\vdash$ s set $\longrightarrow$
[wf**]** [·] H; w: $x \in$s { $y \in$s s | $P[y]$ }; $J[w]$; z: set$\vdash$ <$P[z]$> $\longrightarrow$
[wf**]** [*ext*] H; w: $x \in$s { $y \in$s s | $P[y]$ }; $J[w]\vdash$ $\forall z.P[z]$ fun_prop $\longrightarrow$
[main**]** [*ext*]
1. H
2. w: $x \in$s { $y \in$s s | $P[y]$ }
3. $J[w]$
4. u: $x \in$s s
5. v: $P[x]$
$\vdash$ $T[w]$ $\longrightarrow$

[*ext*] H; w: $x \in$s { $y \in$s s | $P[y]$ }; $J[w]\vdash$ $T[w]$

Functionality

The separation constructor is functional in both the set argument and the proposition.

[16,2528]* **rule `sep_fun` **{|** intro_resource [] **|}** H u v:
[·] H; u: set; v: set$\vdash$ <$P[u; v]$> $\longrightarrow$
[*ext*] H; u: set$\vdash$ $\forall z.P[z; u]$ fun_prop $\longrightarrow$
[*ext*] H; u: set$\vdash$ $\forall z.P[u; z]$ fun_prop $\longrightarrow$
[*ext*] $H\vdash$ $\forall z.s[z]$ fun_set $\longrightarrow$
[*ext*] $H\vdash$ $\forall z.\{ x \in$s $s[z]$ | $P[x; z]$ } fun_set

15.12 Czf_itt_empty module

The Czf_itt_empty module defines an empty set as the set $collect(x \in Void.x)$. Since the *Void* type is empty, the set has no elements.

15.12.1 Parents

include Czf_itt_member

15.12.2 Terms

declare empty = Czf_itt_empty!empty

15.12.3 Rewrites

The empty set uses the empty index type *Void*.

! **rewrite** unfold_empty : $\{\} \longleftrightarrow$ (collect x: Void. x)

15.12.4 Rules

There are only two rules for the empty set: it is a well-formed set, and it has no elements.

[1,14]* **rule empty_isset **{|** intro_resource [] **|}** H:
[*ext*] $H \vdash \{\}$ set
[1,25]* **rule empty_member_elim **{|** elim_resource [] **|}** H J:
[*ext*] H; x: $y \in$s $\{\}$; $J[x] \vdash T[x]$

15.13 Czf_itt_singleton module

The Czf_itt_singleton module defines the singleton sets $\{s\}$, which is a set that contains the single element s. The singleton is used as a building block for pairing, defined in the Czf_itt_pair module.

15.13.1 Parents

include Czf_itt_member

15.13.2 Terms

declare `sing` = Czf_itt_singleton!sing{'s}

15.13.3 Rewrites

The singleton set $\{s\}$ is defined as a set with an index type *Unit*, and an element function that returns the set s. Note that *any* non-empty type can be used as the index type.

! **rewrite** `unfold_sing` : $\{s\} \longleftrightarrow$ (collect x: Unit. s)

15.13.4 Rules

Well-formedness

The singleton $\{s\}$ is well-formed if s is a set.

[1,21]* **rule `sing_isset` **{|** intro_resource [] **|}** H:
[·] $H \vdash s$ set $\longrightarrow$
[*ext*] $H \vdash \{s\}$ set

Introduction and elimination

The *only* element of the singleton set $\{s\}$ is the set s.

[4,33]* **rule `sing_member_elim` **{|** elim_resource [] **|}** H J:
[*ext*] H; x: $y \in$s $\{s\}$; $J[x]$; w: eq(y; s)$\vdash$ $T[x]$ $\longrightarrow$
[*ext*] H; x: $y \in$s $\{s\}$; $J[x] \vdash$ $T[x]$
[2,60]* **rule `sing_member_intro` **{|** intro_resource [] **|}** H:
[wf**]** [·] $H \vdash$ *s1* set $\longrightarrow$
[wf**]** [·] $H \vdash$ *s2* set $\longrightarrow$
[*ext*] $H \vdash$ eq(*s1*; *s2*) $\longrightarrow$
[*ext*] $H \vdash$ *s1* $\in$s $\{$*s2*$\}$

Functionality

The singleton is functional in it's set argument.

[13,396]* **rule `sing_fun` **{|** intro_resource [] **|}** H:
[*ext*] $H \vdash \forall z.s[z]$ fun_set $\longrightarrow$
[*ext*] $H \vdash \forall z.\{s[z]\}$ fun_set

15.14 Czf_itt_union module

The `Czf_itt_union` module gives two definitions of set unions. The $s_1 \cup s_2$ is the binary union of two sets s_1 and s_2. The elements of the binary union include the elements in s_1 as well as the elements of s_2. The general union $\bigcup s$ represents the union of all the elements in s. A set x is a member of the union $\bigcup s$ if it is a member of any element of s.

The binary union is defined for sets $s_1 = \mathit{collect}(x_1 \in T_1.f_1[x_1])$ and $s_2 = \mathit{collect}(x_2 \in T_2.f_2[x_2])$ as a set with the disjoint union index type $T_1 + T_2$ defined in the `Itt_union` module.

$$\mathit{collect}(x_1 \in T_1.f_1[x_1]) \cup \mathit{collect}(x_1 \in T_1.f_1[x_1]) \equiv \mathit{collect}(x \in T_1 + T_2.\textbf{match}\, x\ \textbf{with}\ \mathit{inl}(u) \rightarrow f_1[u] \mid \mathit{inr}(v) \rightarrow f_2[v])$$

The definition of the general union $\bigcup s$ is more difficult. First, suppose the set s has the form $\mathit{collect}(x \in T.f[x])$, and that the elements of s have the form $\mathit{collect}(y \in S_x.g_x[y])$ for each term x in the index type T. The index type of the union is formed as the dependent product space $x{:}T \times S_x$. Given a pair (x, y) in the index type, the elements are $g_x[y]$. Effectively, the union has the following definition.

$$\bigcup \mathit{collect}(x \in T.\lambda x.\mathit{collect}(y \in S[x].g[x;y])) \equiv \mathit{collect}((x,y) \in z{:}T \times S[z].g[x;y])$$

15.14.1 Parents

include Czf_itt_dexists
include Czf_itt_subset

15.14.2 Terms

declare `union` = Czf_itt_union!union{'s1; 's2}

15.14.3 Rewrites

The binary union $s_1 \cup s_2$ is defined by simultaneous induction on the set arguments. The index type of the result is the disjoint union of their index types.

*! **rewrite** unfold_bunion :*
(*s1* ∪s *s2*) ⟷
(let set(*a1*, *f1*).*g1* = *s1* in
let set(*a2*, *f2*).*g2* = *s2* in
collect *x*: (*a1* + *a2*).
match *x* with inl *z* -> (*f1 z*) | inr *z* -> (*f2 z*))

[1,136] **rewrite reduce_bunion :*
(collect *x1*: *t1*. *f1*[*x1*] ∪s collect *x2*: *t2*. *f2*[*x2*]) ⟷
(collect *x*: (*t1* + *t2*).
match *x* with inl *z* -> *f1*[*z*] | inr *z* -> *f2*[*z*])

The general union is formed by a nested induction on the set argument. This term formalizes the informal discussion above.

*! **rewrite** unfold_union :*
(∪s *s*) ⟷
(let set(*a1*, *f1*).*g1* = *s* in
collect *y*: (*x*:*a1* × (let set(*a2*, *f2*).*g2* = *f1 x* in *a2*)).
(**match** *y* **with**
(*u*,*v*) →
(let set(*a3*, *f3*).*g3* = *f1 u* in *f3 v*)))

15.14.4 Rules

Well-formedness

Both forms of union are well-formed if their arguments are sets.

[8,221] **rule bunion_isset {|* intro_resource [] *|}* H:
[wf**]** [·] *H*⊢ *s1* set ⟶
[wf**]** [·] *H*⊢ *s2* set ⟶
[*ext*] *H*⊢ (*s1* ∪s *s2*) set
[15,299] **rule union_isset {|* intro_resource [] *|}* H:
[wf**]** [·] *H*⊢ *s1* set ⟶
[*ext*] *H*⊢ (∪s *s1*) set

Introduction

The binary union $s_1 \cup s_2$ has two membership introduction forms for an argument set x; the set x may be a member of s_1 or it may be a member of s_2.

[10,425]* **rule `bunion_member_intro_left`
{| intro_resource [(SelectOption 1)] **|}** H:
[wf**]** [·] $H \vdash x$ set $\longrightarrow$
[wf**]** [·] $H \vdash s1$ set $\longrightarrow$
[wf**]** [·] $H \vdash s2$ set $\longrightarrow$
[*ext*] $H \vdash x \in \mathrm{s}\ s1 \longrightarrow$
[*ext*] $H \vdash x \in \mathrm{s}\ (s1 \cup \mathrm{s}\ s2)$
[10,425]* **rule `bunion_member_intro_right`
{| intro_resource [(SelectOption 2)] **|}** H:
[wf**]** [·] $H \vdash x$ set $\longrightarrow$
[wf**]** [·] $H \vdash s1$ set $\longrightarrow$
[wf**]** [·] $H \vdash s2$ set $\longrightarrow$
[*ext*] $H \vdash x \in \mathrm{s}\ s2 \longrightarrow$
[*ext*] $H \vdash x \in \mathrm{s}\ (s1 \cup \mathrm{s}\ s2)$

A set x is in the general union $\bigcup s$ if there is some element $y \in_s s$ for which $x \in_s y$.

[42,1042]* **rule `union_member_intro` **{|** intro_resource [] **|}** H:
[wf**]** [·] $H \vdash x$ set $\longrightarrow$
[wf**]** [·] $H \vdash s$ set $\longrightarrow$
[*ext*] $H \vdash \exists y \in \mathrm{s}\ s.x \in \mathrm{s}\ y \longrightarrow$
[*ext*] $H \vdash x \in \mathrm{s}\ (\cup \mathrm{s}\ s)$

Elimination

The elimination form for membership in the binary union performs a case analysis on membership in the two sets in the binary union.

[20,710]* **rule `bunion_member_elim` **{|** elim_resource [] **|}** H J:
[wf**]** [·] H; x: $y \in \mathrm{s}\ (s1 \cup \mathrm{s}\ s2)$; $J[x] \vdash y$ set $\longrightarrow$
[wf**]** [·] H; x: $y \in \mathrm{s}\ (s1 \cup \mathrm{s}\ s2)$; $J[x] \vdash s1$ set $\longrightarrow$
[wf**]** [·] H; x: $y \in \mathrm{s}\ (s1 \cup \mathrm{s}\ s2)$; $J[x] \vdash s2$ set $\longrightarrow$
[*ext*] H; x: $y \in \mathrm{s}\ (s1 \cup \mathrm{s}\ s2)$; $J[x]$; z: $y \in \mathrm{s}\ s1 \vdash T[x] \longrightarrow$
[*ext*] H; x: $y \in \mathrm{s}\ (s1 \cup \mathrm{s}\ s2)$; $J[x]$; z: $y \in \mathrm{s}\ s2 \vdash T[x] \longrightarrow$
[*ext*] H; x: $y \in \mathrm{s}\ (s1 \cup \mathrm{s}\ s2)$; $J[x] \vdash T[x]$

The elimination form for the general union $x \in_s \bigcup s$ produces a witness $z \in_s y$ for which $x \in_s z$.

[28,664]* **rule `union_member_elim` {| elim_resource [] |} H J:
[wf**]** [·] H; x: y ∈s (∪s s); $J[x]$⊢ y set ⟶
[wf**]** [·] H; x: y ∈s (∪s s); $J[x]$⊢ s set ⟶
[*ext*] H; x: y ∈s (∪s s); $J[x]$; z: set; u: z ∈s s; v: y ∈s z⊢ $T[x]$ ⟶
[*ext*] H; x: y ∈s (∪s s); $J[x]$⊢ $T[x]$

The union types are both functional in their arguments.

[18,920]* **rule `bunion_fun` {| intro_resource [] |} H:
[*ext*] H⊢ ∀z.*s1*[z] fun_set ⟶
[*ext*] H⊢ ∀z.*s2*[z] fun_set ⟶
[*ext*] H⊢ ∀z.(*s1*[z] ∪s *s2*[z]) fun_set
[15,585]* **rule `union_fun` {| intro_resource [] |} H:
[*ext*] H⊢ ∀z.s[z] fun_set ⟶
[*ext*] H⊢ ∀z.(∪s s[z]) fun_set

15.15 Czf_itt_pair module

The `Czf_itt_pair` module defines the binary pairing constructor (s_1, s_2). The pair is derived from the union and singleton.

$$(s_1, s_2) \equiv \{s_1\} \cup \{s_2\}$$

The pair has two elements: the set s_1 and the set s_2.

15.15.1 Parents

include Czf_itt_union
include Czf_itt_singleton

15.15.2 Terms

declare `pair` = Czf_itt_pair!pair{'s1; 's2}

15.15.3 Rewrites

The pair is a union of two singleton sets.

! **rewrite** `unfold_pair` : (*s1*, *s2*) ⟷ ({*s1*} ∪s {*s2*})

15.15.4 Rules

Well-formedness

The pair is a set if both arguments are sets.

[1,19]* **rule `pair_isset` **{|** intro_resource [] **|}** H:
[wf**]** [·] $H\vdash$ *s1* set $\longrightarrow$
[wf**]** [·] $H\vdash$ *s2* set $\longrightarrow$
[*ext*] $H\vdash$ (*s1*, *s2*) set

Introduction

The elements of the pair (s_1, s_2) are the sets s_1 and s_2.

[2,44]* **rule `pair_member_intro_left`
{| intro_resource [(SelectOption 1)] **|}** H:
[wf**]** [·] $H\vdash$ *x* set $\longrightarrow$
[wf**]** [·] $H\vdash$ *s1* set $\longrightarrow$
[wf**]** [·] $H\vdash$ *s2* set $\longrightarrow$
[*ext*] $H\vdash$ eq(*x*; *s1*) $\longrightarrow$
[*ext*] $H\vdash$ *x* $\in$s (*s1*, *s2*)
[2,44]* **rule `pair_member_intro_right`
{| intro_resource [(SelectOption 2)] **|}** H:
[wf**]** [·] $H\vdash$ *x* set $\longrightarrow$
[wf**]** [·] $H\vdash$ *s1* set $\longrightarrow$
[wf**]** [·] $H\vdash$ *s2* set $\longrightarrow$
[*ext*] $H\vdash$ eq(*x*; *s2*) $\longrightarrow$
[*ext*] $H\vdash$ *x* $\in$s (*s1*, *s2*)

Elimination

The *only* elements of the pair (s_1, s_2) are the sets s_1 and s_2.

[10,134]* **rule `pair_member_elim` **{|** elim_resource [] **|}** H J:
[wf**]** [·] H; x: y $\in$s (*s1*, *s2*); $J[x]\vdash$ y set $\longrightarrow$
[wf**]** [·] H; x: y $\in$s (*s1*, *s2*); $J[x]\vdash$ *s1* set $\longrightarrow$
[wf**]** [·] H; x: y $\in$s (*s1*, *s2*); $J[x]\vdash$ *s2* set $\longrightarrow$
[*ext*] H; x: y $\in$s (*s1*, *s2*); $J[x]$; z: eq(y; *s1*)$\vdash$ $T[x]$ $\longrightarrow$
[*ext*] H; x: y $\in$s (*s1*, *s2*); $J[x]$; z: eq(y; *s2*)$\vdash$ $T[x]$ $\longrightarrow$
[*ext*] H; x: y $\in$s (*s1*, *s2*); $J[x]\vdash$ $T[x]$

Functionality

The pair is functional in both its arguments.

[1,19]* **rule `pair_fun` **{|** intro_resource [] **|}** H:
[*ext*] $H \vdash \forall z.s1[z]$ fun_set $\longrightarrow$
[*ext*] $H \vdash \forall z.s2[z]$ fun_set $\longrightarrow$
[*ext*] $H \vdash \forall z.(s1[z], s2[z])$ fun_set

15.16 Czf_itt_nat module

The `Czf_itt_nat` module defines the infinite set ω. We use the definition of ω as the definition of the natural numbers. The `zero` term represents the zero, and the `succ` term $s(i)$ represents $i + 1$. The construction is the usual ordinal construction.

$$\begin{array}{lcl} 0_s & \equiv & \{\} \\ s(0_s) & \equiv & 0_s \cup \{0_s\} \\ & \vdots & \\ s(i) & \equiv & i \cup \{i\} \end{array}$$

The definition of the ω set uses the `list` type as an index type, and it codes the elements using the list induction combinator.

$$\omega \equiv \mathit{collect}(l \in \mathit{list}(\mathit{Unit}).\textbf{match}\, l\, \textbf{with}\, \mathit{cons}(h,t).g \rightarrow s(g))$$

We also define the usual ordering $i <_s j$ on numbers (this is just a membership judgment).

15.16.1 Parents

include Czf_itt_member
include Czf_itt_singleton
include Czf_itt_union
include Czf_itt_empty
include Czf_itt_implies

15.16.2 Terms

declare `inf` = Czf_itt_nat!inf

declare zero = Czf_itt_nat!zero
declare succ = Czf_itt_nat!succ{'i}
declare lt = Czf_itt_nat!lt{'i; 'j}

15.16.3 Rewrites

The following four rewrites give the definition of the natural numbers. The ω set itself is an ordinal construction.

! **rewrite** unfold_zero : 0s $\longleftrightarrow$ {}
! **rewrite** unfold_succ : s(i) $\longleftrightarrow$ (i $\cup$s {i})
! **rewrite** unfold_inf :
    $\mathbb{N}$s $\longleftrightarrow$
    (collect l: Unit List.
        match l with [] -> {} | h::t.g -> s(g))

! **rewrite** unfold_lt : (i < j) $\longleftrightarrow$ (i $\in$s j)

15.16.4 Rules

Well-formedness

Zero is a set, and the successor of *any* set is a set. Infinity is also a set.

[1,7]* **rule zero_isset **{|** intro_resource [] **|}** H:
    [*ext*] $H\vdash$ 0s set
[1,16]* **rule succ_isset **{|** intro_resource [] **|}** H:
    [$\cdot$] $H\vdash$ i set $\longrightarrow$
    [*ext*] $H\vdash$ s(i) set
[4,63]* **rule inf_isset **{|** intro_resource [] **|}** H:
    [*ext*] $H\vdash$ $\mathbb{N}$s set
[1,16]* **rule succ_fun **{|** intro_resource [] **|}** H:
    [*ext*] $H\vdash$ $\forall z.s[z]$ fun_set $\longrightarrow$
    [*ext*] $H\vdash$ $\forall z.$s($s[z]$) fun_set

The zero is also a number, and the successor is a number if it's argument is a number.

[3,152]* **rule zero_isnat **{|** intro_resource [] **|}** H:
    [*ext*] $H\vdash$ 0s $\in$s $\mathbb{N}$s
[16,398]* **rule succ_isnat **{|** intro_resource [] **|}** H:
    [wf**]** [*ext*] $H\vdash$ i set $\longrightarrow$
    [*ext*] $H\vdash$ i $\in$s $\mathbb{N}$s $\longrightarrow$
    [*ext*] $H\vdash$ s(i) $\in$s $\mathbb{N}$s

Induction

The induction rule performs induction on an arbitrary number expression in the goal. The goal must be functional over *all* sets. The induction has the usual cases: to prove $C[i]$ for a number i, prove $C[0_s]$, and given $C[i]$, prove $C[s(i)]$. The proof of this rule is rather extensive, but it is derived from usual inductive reasoning on sets.

[62,1333]* **rule `nat_elim` H bind(z.C[z]) i:
[wf**]** [*ext*] $H\vdash$ i ∈S ℕs ⟶
[wf**]** [*ext*] $H\vdash$ ∀z.C[z] fun_prop ⟶
[base**]** [*ext*] $H\vdash$ C[0s] ⟶
[step**]** [*ext*] H; j: set; u: j ∈s ℕs; v: C[j]$\vdash$ C[s(j)] ⟶
[*ext*] $H\vdash$ C[i]

Functionality

The $i <_s j$ relation is functional on its arguments, and it is also a restricted proposition.

[1,13]* **rule `lt_fun` **{|** intro_resource [] **|}** H:
[*ext*] $H\vdash$ ∀z.$s1$[z] fun_set ⟶
[*ext*] $H\vdash$ ∀z.$s2$[z] fun_set ⟶
[*ext*] $H\vdash$ ∀z.($s1$[z] < $s2$[z]) fun_prop
[1,13]* **rule `lt_restricted` **{|** intro_resource [] **|}** H:
[wf**]** [·] $H\vdash$ $s1$ set ⟶
[wf**]** [·] $H\vdash$ $s2$ set ⟶
[*ext*] $H\vdash$ <($s1$ < $s2$)>

Zero

The following three rules characterize the `zero` set. Zero is a number that is smaller than every successor.

The rules play a crucial rule in forcing the ω set to be infinite. Induction by itself is not sufficient, because the induction rule is valid if all the numbers are equal. The zero rules state that the zero term, at least, is different from any successor.

[6,135]* **rule `zero_member_intro` **{|** intro_resource [] **|}** H:
[wf**]** [*ext*] $H\vdash$ i set ⟶
[wf**]** [*ext*] $H\vdash$ i ∈s ℕs ⟶
[*ext*] $H\vdash$ 0s < s(i)
[1,7]* **rule `zero_member_elim` **{|** elim_resource [] **|}** H J:
[wf**]** [·] H; x: i < 0s; J[x]$\vdash$ i ∈s ℕs ⟶
[*ext*] H; x: i < 0s; J[x]$\vdash$ T[x]

Successor

The following two rules characterize the successor. The relation $s(i) <_s s(j)$ is true if-and-only-if $i <_s j$. With these final rules, we can show that $i \neq s(i)$, so the set ω is actually infinite.

[14,666]* **rule succ_member_intro1 {| intro_resource [] |} H:
[wf**]** [·] $H \vdash i$ set $\longrightarrow$
[wf**]** [·] $H \vdash j$ set $\longrightarrow$
[wf**]** [*ext*] $H \vdash i \in_s \mathbb{N}_s$ $\longrightarrow$
[wf**]** [*ext*] $H \vdash j \in_s \mathbb{N}_s$ $\longrightarrow$
[*ext*] $H \vdash i < j$ $\longrightarrow$
[*ext*] $H \vdash s(i) < s(j)$
[28,1296]* **rule succ_member_elim1 {| elim_resource [] |} H J:
[wf**]** [·] H; x: $s(i) < s(j)$; $J[x] \vdash i$ set $\longrightarrow$
[wf**]** [·] H; x: $s(i) < s(j)$; $J[x] \vdash j$ set $\longrightarrow$
[wf**]** [*ext*] H; x: $s(i) < s(j)$; $J[x] \vdash j \in_s \mathbb{N}_s$ $\longrightarrow$
[*ext*] H; x: $s(i) < s(j)$; $J[x]$; w: $i < j \vdash T[x]$ $\longrightarrow$
[*ext*] H; x: $s(i) < s(j)$; $J[x] \vdash T[x]$

15.16.5 Tactics

natIndT The (natIndT t) tactic performs induction on the expression t, which must be a well-formed number.

15.17 Czf_itt_rel module

The Czf_itt_rel module defines Aczel's *collection* scheme :

$$\mathbb{B}(x \in a, y \in b.P[x;y]) \equiv (\forall x \in_s a.\exists y \in_s b.\phi \wedge \forall y \in_s b.\exists x \in_s a.\phi.$$

There are no rules in this module, except for well-formedness. The rel term is just a definition.

15.17.1 Parents

include Czf_itt_dall
include Czf_itt_dexists

15.17.2 Terms

declare `rel` = Czf_itt_rel!rel{a, b. P['a; 'b]; 's1; 's2}

15.17.3 Rewrites

! **rewrite** `unfold_rel` :
(𝔹$a \in s1$, $b \in s2$. $P[a; b]$) ⟷
((∀$x \in$s $s1$.∃$y \in$s $s2$.$P[x; y]$) ∧ (∀$y \in$s $s2$.∃$x \in$s $s1$.$P[x; y]$))

15.17.4 Rules

The `rel` term is well-formed if the the proposition P is well-formed, and if the arguments s_1 and s_2 are sets.

[1,131]* **rule `rel_type` **{|** intro_resource [] **|}** H u v:
[·] H; u: set; v: set⊢ $P[u; v]$ Type ⟶
[·] H⊢ $s1$ set ⟶
[·] H⊢ $s2$ set ⟶
[ext] H⊢ (𝔹$x \in s1$, $y \in s2$. $P[x; y]$) Type

15.18 Czf_itt_power module

The `Czf_itt_power` module defines *subset* collection. Contrary to the name, the subset collection is *not* a power set, but it has some similarities. The subset collection constructor has the form $\mathbb{P}s_1 \to s_1$, where s_1 and s_2 are sets. If the the canonical forms are $s_1 = collect(x_1 \in T_1.f_1[x_1])$ and $s_2 = collect(x_2 \in T_2.f_2[x_2])$, the elements of the power set are the subsets of s_2 that are defined by the images of the computable functions in the space $T_1 \to T_2$.

$$\mathbb{P}s_1 \to s_1 \equiv collect(g \in T_1 \to T_2.collect(x \in T_1.f_2[g(x)]))$$

There is only one significant rule in this module: the axiom of subset collection.

15.18.1 Parents

include Czf_itt_subset
include Czf_itt_rel

15.18.2 Terms

declare power = Czf_itt_power!power{'s1; 's2}

15.18.3 Rewrites

The subset collection is defined by simultaneous induction on the two set arguments.

! **rewrite** unfold_scoll :
ℙ(*s1* → *s2*) ⟷
(let set(*T1*, *f1*).*g1* = *s1* in
let set(*T2*, *f2*).*g2* = *s2* in
collect *x*: (*T1* → *T2*). collect *y*: *T1*. *f2* (*x* *y*))

[1,134]* **rewrite reduce_scoll :
ℙ((collect *x1*: *T1*. *f1*[*x1*]) → (collect *x2*: *T2*. *f2*[*x2*])) ⟷
(collect *x*: (*T1* → *T2*). collect *y*: *T1*. *f2*[(*x* *y*)])

15.18.4 Rules

Well-formedness

The subset collection is well-formed if its arguments are sets.

[7,180]* **rule power_isset1 **{|** intro_resource [] **|}** H:
[wf**]** [·] *H*⊢ *s1* set ⟶
[wf**]** [·] *H*⊢ *s2* set ⟶
[*ext*] *H*⊢ ℙ(*s1* → *s2*) set

The subset collection axiom

There is an element of the power set for each computable function $s_1 \to s_2$.

[57,4037]* **rule power_thm H bind(*x*.bind(*y*.*P*[*x*; *y*])) *a* *b*:
[wf**]** [·] *H*⊢ *a* set ⟶
[wf**]** [·] *H*⊢ *b* set ⟶
[wf**]** [·] *H*; x: set; y: set⊢ *P*[*x*; *y*] Type ⟶

[wf**]** [*ext*] H; x: set$\vdash$ $\forall z.P[x; z]$ fun_prop $\longrightarrow$
[wf**]** [*ext*] H; x: set$\vdash$ $\forall z.P[z; x]$ fun_prop $\longrightarrow$
[antecedent**]** [*ext*] H; x: set; u: $x \in$s $a\vdash$ $\exists y \in$s $b.P[x; y]$ $\longrightarrow$
[*ext*] H; z: set; u: $z \in$s $\mathbb{P}(a \to b)$; v: $\mathbb{B}x \in a, y \in z.\ P[x; y]\vdash$ C $\longrightarrow$
[*ext*] $H\vdash$ C

15.18.5 Tactics

`powerT` The (`powerT` t) tactic applies the `power_thm` rule. The argument t is a `rel` term $\mathbb{B}(x \in a, y \in b.P[x, y])$ that specifies the proposition P and the sets a and b that are the arguments to the `power_thm`.

15.19 Czf_itt_axioms module

The `Czf_itt_axioms` defines the remaining axioms of the set theory as axioms. This includes the set induction scheme, and the strong collection.

15.19.1 Parents

The `Czf_itt_axiom` module includes the entire logical part of the theory, as well as the definition of the `rel` (for use in the definition of the strong collection theorem).

include Czf_itt_true
include Czf_itt_false
include Czf_itt_and
include Czf_itt_or
include Czf_itt_implies
include Czf_itt_all
include Czf_itt_exists
include Czf_itt_sall
include Czf_itt_sexists
include Czf_itt_dall
include Czf_itt_dexists
include Czf_itt_rel

15.19.2 Rules

Set induction

The set induction formalizes the induction scheme. A goal $P[z]$ can be proven for a set z if it can be proved for an arbitrary set x, given that it holds on all the elements.

The proof of induction follows directly from W-type induction.

[14,320]* **rule `set_induction` H x w:
[*ext*] H; x: set$\vdash$ $P[x]$ Type $\longrightarrow$
[*ext*] $H\vdash$ $\forall z.P[z]$ fun_prop $\longrightarrow$
[*ext*] H; x: set; w: $\forall z \in$s $x.P[z]\vdash$ $P[x]$ $\longrightarrow$
[*ext*] $H\vdash$ $\forall_s z.P[z]$

Strong Collection

The strong collection axioms states that for every proof of a for-all/exists formula $\forall x \in_s s_1.\exists_s y{:}P[x.y]$, there is a set s_3 that contains the collection of sets that were chosen by the existential. The reason that the collection is not defined as a set constructor is the the proof of the for-all/exists formula is part of the construction. If the set s_1 has canonical for $s_1 = \mathit{collect}(x \in T.f[x])$, the proof provides a witness that inhabits the function space $T \to \mathit{set}$. The canonical form of the proof is a `lambda`-function $\lambda x.s_x$, which can be used to form the set collection $\mathit{collect}(x \in T.s_x)$.

[37,1545]* **rule `collection` H *s1* bind(x.bind(y.$P[x; y]$)) s2 x y w:
[$\cdot$] $H\vdash$ *s1* set $\longrightarrow$
[$\cdot$] H; x: set; y: set$\vdash$ $P[x; y]$ Type $\longrightarrow$
[*ext*] $H\vdash$ $\forall x \in$s *s1*.$\exists_s y.P[x; y]$ $\longrightarrow$
[*ext*] H; s2: set; w: $\mathbb{B}x \in s1, y \in s2.\ P[x; y]\vdash$ C $\longrightarrow$
[*ext*] $H\vdash$ C

Subset collection

The `Czf_itt_power` module defines the subset collection set constructor $\mathbb{P}s_1 \to s_1$. For completeness, we reprove the axiom form of the subset collection.

[44,2291]* **rule `subset_collection`
H a b c bind(u.bind(x.bind(y.$P[u; x; y]$))):
[*ext*] $H\vdash$ a set $\longrightarrow$
[*ext*] $H\vdash$ b set $\longrightarrow$
[$\cdot$] H; u: set; x: set; y: set$\vdash$ $P[u; x; y]$ Type $\longrightarrow$

[*ext*] H; u: set; x: set$\vdash$ $\forall y.P[u; x; y]$ fun_prop $\longrightarrow$
[*ext*]
1. H
2. w:
$\exists_s c.\forall_s u.((\forall x \in \mathrm{s}\ a.\exists y \in \mathrm{s}\ b.P[u; x; y]) \rightarrow$
$\vdash C \longrightarrow$
[*ext*] $H\vdash C$

BIBLIOGRAPHY

[AC91] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. In *18th Annual ACM Symposium on Principles of Programming Languages*, pages 104–118. ACM, January 1991. 10.6

[AC94a] Martín Abadi and Luca Cardelli. A theory of primitive objects: Second–order systems. In *Proceedings of European Symposium on Programming*, pages 1–24. Springer, April 1994. 10.6

[AC94b] Martín Abadi and Luca Cardelli. A theory of primitive objects: Untyped and first–order systems. *Theoretical Aspects of Computer Software*, pages 296–320, April 1994. 10.1, 10.4.1, 10.6

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996. 11.1, 11.2

[ACE+00] Stuart Allen, Robert Constable, Richard Eaton, Christoph Kreitz, and Lori Lorigo. The nuprl open logical environment. In D. McAllester, editor, *17th International Conference on Automated Deduction*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 170–176. Springer Verlag, 2000. 1, 7.9

[ACHA90] Stuart F. Allen, Robert L. Constable, Douglas J. Howe, and William Aitken. The semantics of reflected proof. In *Proc. of Fifth Symp. on Logic in Comp. Sci.*, pages 95–197. IEEE, June 1990. 2

[ACV96] Martín Abadi, Luca Cardelli, and Ramesh Viswanathan. An Interpretation of Objects and Object Types. In *ACM Symposium on Principles of Programming Languages*, 1996. 11.1, 11.2, 11.3.1

[Acz86] Peter Aczel. The type theoretic interpretation of constructive set theory. In *Logic, Methodology and Philosophy of Science VII*, pages 17–49. Elsevier Science Publishers, 1986. 15

[Acz97] Peter Aczel. Notes on constructive type theory, May 1997. unpublished manuscript. 15

[AE00] Stuart Allen and Richard Eaton. Display and editing in the Nuprl systems, 2000. private communication. 7.3

[AL95] Martín Abadi and Leslie Lamport. Conjoining specifications. *ACM Toplas*, 17(3), May 1995. 4, 4.7

[All86] Stuart F. Allen. *The Semantics of Type Theoretic Languages*. PhD thesis, Cornell University, August 1986. 5.3, 11.1, 14

[All87] Stuart F. Allen. A non-type-theoretic definition of martin-lof's types. In *Proceedings of the Second Conference on Logic in Computer Science*, pages 215–224, June 1987. 11.1

[Bat79] J. L. Bates. *A Logic for Correct Program Development*. PhD thesis, Cornell University, 1979. 1, 8.2

[BC85] J. L. Bates and Robert L. Constable. Proofs as programs. *ACM Trans. Program. Lang. and Syst.*, 7(1):53–71, 1985. 1

[BC93] David A. Basin and Robert L. Constable. *Metalogical Frameworks*, pages 1–29. Cambridge University Press, 1993. 5.3, 10.6

[BDGB98] Ozalp Babaoglu, Renzo Davoli, L. Giachini, and G. Baker. System support for partition-aware network applications. In *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems*, May 1998. 4.7

[Bir97] Kenneth P. Birman. *Building Secure and Reliable Network Applications*. Manning Publishing Company and Prentice Hall, January 1997. 4.7

[Bis67] E. Bishop. *Foundations of Constructive Analysis*. McGraw Hill, NY, 1967. 15

[BJ87] Kenneth P. Birman and Thomas A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proc 11th Symposium on Operating Systems Principles (SOSP)*, pages 123–138, November 1987. 4.7

[BK91] David A. Basin and M. Kaufmann. The Boyer-Moore prover and Nuprl: An experimental comparison. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 89–119. Cambridge University Press, 1991. 8.7

[BM79] R. S. Boyer and J. S. Moore. *A Computational Logic*. Academic Press, New York, 1979. 8.7

[BM94] Maria Paola Bonacina and William Mccune. Distributed theorem proving by peers. In *1194 Conference on Automated Deduction (CADE12)*, pages 841–845, 1994. 9.7

[Bre91] R. Breu. *Algebraic Specification Techniques in Object Oriented Programming Environments*. Springer–Verlag, 1991. 10.6

[Bru94] Kim B. Bruce. A paradigmatic object-oriented programming language: Design, static typing, and semantics. *Journal of Functional Programming*, 4(2):127–206, April 1994. 11.1

[CAB+86] Robert L. Constable, Stuart F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, Douglas J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, NJ, 1986. 1, 2.5, 4.7, 7.3, 8.1, 9.1, 9.2, 10.3, 10.6, 14

[CCM85] G. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. *Functional Programming Languages and Computer Architecture, LNCS 201*, pages 50–64, 1985. 8.4

[CDE+99] Manuel Clavel, Fransisco Durán, Steven Eker, Patrick Lincoln, Narciso Martín-Oliet, José Meseguer, and Jose F. Quesada. The Maude system. In *10th International Conference on Rewriting Techniques and Applications (RTA'99)*, pages 240–243. Springer, 1999. LNCS 1631. 5.3

[CKB84] Robert L. Constable, T. Knoblock, and J.L. Bates. Writing programs that construct proofs. *J. Automated Reasoning*, 1(3):285–326, 1984. 8.5

[CNSvS94] Thierry Coquand, Bengt Nordström, Jan M. Smith, and Björn von Sydow. Type Theory and Programming. *EATCS*, February 1994. bulletin no 52. 8.1

[CO78] Robert L. Constable and Michael J. O'Donnell. *A Programming Logic*. Winthrop, Mass., 1978. 1

[Con71] Robert L. Constable. Constructive mathematics and automatic program writers. In *Proceedings of the IFIP Congr.*, pages 229–233. North-Holland, 1971. 1

[Con92] Robert L. Constable. Lectures on: Classical proofs as programs. *Constructive Methods of Computing Science,* NATO ASI Series, 1992. 14, 14.3.2

[Con98] Robert L. Constable. *Handbook of Proof Theory*, chapter X: Types in Logic, Mathematics and Programming, pages 650–757. Elsevier Science B.V., 1998. 14, 14.3.2

[Con00] Robert Constable. The structure of Nuprl's type theory, 2000. www.cs.cornell.edu/Info/Projects/NuPrl. 14, 14.3.2

[COR+95] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A Tutorial Introduction to PVS. In *WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques*, April 1995. `http://www.csl.sri.com/sri-csl-fm.html`. 8.1

[CP96] Adriana B. Compagnoni and Benjamin C. Pierce. Intersection types and multiple inheritance. *Mathematical Structures in Computer Science*, 1996. 11.5

[Crang] Karl Crary. Recursive computation in foundational type theory. Technical report, Department of Computer Science, Cornell University, Forthcoming. 11.3.3

[CV98] D.J. Cook and R.C. Varnell. Adaptive parallel iterative deepening search. *Journal of Artificial Intelligence Research*, 9:139–165, 1998. 9.7

[DB] H. De Bruijn. Telescope mappings. unpublished manuscript M15. 10.3

[deB70] N. G. deBruijn. The mathematical language Automath: Its usage and some of its extensions. In J. P. Seldin and J. R. Hindley, editors, *Symposium on Automatic Demonstration,* Lecture Notes in Mathematics, Vol. 125, pages 29–61. Springer-Verlag, 1970. 5.3

[deB80] N. G. deBruijn. A survey of the project Automath. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606. Academic Press, 1980. 5.3

[DF97] J. Denzinger and F. Fuchs. Cooperation in theorem proving by loosely coupled heuristics. Technical Report SR-97-04, University of Kaiserslautern, 1997. 9.7

[DFH+93] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chet Murthy, Catherine Parent, Christine Paulin-Mohring, and Benjamin Werner. The coq proof assistant user's guide. Technical Report Rapport Techniques 154, INRIA, Rocquencourt, France, 1993. Version 5.8. 2.5, 8.1, 9.1, 9.2

[DP99] R. Dyckhoff and L. Pinto. Proof search in constructive logics. In S. Cooper and J. Truss, editors, *Proceedings Logic Colloquium 97, i.e. Sets and Proofs*, pages 53–65, 1999. 13.1.9

[dR] D. de Rauglaudre. The camlp4 reference manual. available at `http://caml.inria.fr/camlp4/manual/index.html`. 7.2

[Dyc92] R. Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. In *The Journal of Symbolic Logic*, pages Vol. 57, Number 3, September 1992. 4

[Fek93] Alan Fekete. Formal models of communications services: A case study. *IEEE Computer*, 26(8):37–47, August 1993. 4.7

[FLR98] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, 1998. 9.7

[FLS97] Alan Fekete, Nancy Lynch, and Alex Shvartsman. Specifying and using partitionable group communication service. In *Proc.16thAnnual ACM Symposium on Principles of Dist. Comp.*, pages 52–62, 1997. 4.7

[Gir72] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'a rithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972. 11.1

[GM93] M.J.C. Gordon and T.F. Melham. *Introduction to HOL*. Cambridge University Press, 1993. 8.1, 9.1, 10.3

[GM94] Carl A. Gunter and John C. Mitchell, editors. *Theoretical Aspects of Object–Oriented Programming*. MIT Press, 1994. 10.6, 11.1

[GMW79] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78. Springer Verlag, 1979. Lecture Notes in Computer Science. 2, 4

[Har96] John Harrison. HOL Light: A tutorial introduction. In *Formal Methods in Computer-Aided Design (FMCAD'96)*, pages 265–269. Springer LNCS 1166, 1996. 8.7

[Hay97] Mark G. Hayden. *The Ensemble System*. PhD thesis, Dept. of Computer Science, Cornell University, January 1997. 1.3, 4, 9.7

[Hay98] Mark Hayden. The Ensemble system. Technical Report TR98-1662, Cornell University, 1998. 1.3, 4, 9.7

[HHP93] Rober Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1), January 1993. 1, 1.3, 5.2, 5.3

[Hic96] Jason J. Hickey. Formal objects in type theory using very dependent types. In *Foundations of Object Oriented Languages 3*, 1996. Available electronically through the FOOL 3 home page at Williams College. 4.7, 11.1, 11.5

[Hic97] Jason J. Hickey. Nuprl-Light: An implementation framework for higher–order logics. In *14th International Conference on Automated Deduction*. Springer, 1997. 8.1, 11.1

[Hic99] Jason Hickey. Fault-tolerant distributed theorem proving. In Harald Ganzinger, editor, *Automated deduction – CADE-16, 16th International Conference on Automated Deduction*, LNAI 1632, pages 227–231, Trento, Italy, July 7–10, 1999. Springer-Verlag.

[HL94] Robert Harper and Mark Lillibridge. A type–theoretic approach to higher–order modules with sharing. In *21st Annual ACM Symposium on Principles of Programming Languages*, pages 123–137. ACM, January 1994. 10.6

[HLvR99] Jason Hickey, Nancy Lynch, and Robbert van Renesse. Specifications and proofs for Ensemble layers. In *TACAS '99*, March 1999. 4.7

[HNSS98] M. Hofmann, W. Naraschewski, M. Steffen, and T. Stroup. Inheritance of proofs. In *TAPOS–Theory and Practice of Object Systems*, volume 4, pages 51–69. Wiley, 1998. 11.1, 11.5

[How96] Douglas J. Howe. Semantic foundations for embedding HOL in Nuprl. In *AMAST '96*, 1996. 11.1

[How98] Douglas J. Howe. A type annotation scheme for Nuprl. In *Theorem Proving in Higher-Order Logics*. Springer, 1998. 8.7

[HP91] Robert Harper and Benjamin Pierce. A record calculus based on symmetric concatenation. In *18th Annual ACM Symposium on Principles of Programming Languages*, pages 131–142. ACM, January 1991. 10.6

[HP95a] Martin Hofmann and Benjamin Pierce. Positive subtyping. In *Proceedings of Twenty-Second Annual ACM Symposium on Principles of Programming Languages*, pages 186–197. ACM, January 1995. 11.5

[HP95b] Martin Hofmann and Benjamin Pierce. A unifying type-theoretic framework for objects. *Journal of Functional Programming*, 5(4):593–635, October 1995. 11.5

[HS94] Douglas J. Howe and Scott D. Stoller. An operational approach to combining classical set theory and functional programming languages. In and J. C. Mitchell M. Hahiya, editor, Lecture Notes in Computer Science, Vol. 789, pages 36–55, New York, April 1994. International Symposium TACS '94, Springer, Berlin. Theoretical Aspects of Computer Software. 1

[Jac94] Paul B. Jackson. *The Nuprl Proof Development System, Version 4.1 Reference Manual and User's Guide*. Cornell University, Ithaca, NY, February 1994. 1, 5.1, 7.3, 7.5, 8.4

[Jac95] Paul Bernard Jackson. *Enhancing the NuPRL Proof Development System and Applying it to Computational Abstract Algebra*. PhD thesis, Cornell University, January 1995. 1, 4.7, 10.6, 11.5

[Jon94] Bengt Jonsson. Compositional specification and verification of distributed systems. *ACM Trans. on Prog. Lang. and Systems*, 16(2):259–303, March 1994. 4.7

[Kar96] David A. Karr. *Protocol Composition on Horus*. PhD thesis, Dept. of Computer Science, Cornell University, December 1996. 4.7

[KHH98] Christoph Kreitz, Mark Hayden, and Jason Hickey. A proof environment for the development of group communication systems. In C. Kirchner and H. Kirchner, editors, *15^{th} International Conference on Automated Deduction*, volume 1421

of *Lecture Notes in Artificial Intelligence*, pages 317–332. Springer Verlag, 1998.

[KM97] Matt Kaufmann and J. Moore. An industrial strength theorem prover for a logic based on common lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, April 1997. 8.7

[Kre97] Christoph Kreitz. Formal reasoning about communication systems I: Embedding ML into type theory. Technical Report TR97-1637, Cornell University. Department of Computer Science, June 1997. 1

[Kre99] Christoph Kreitz. Automated fast-track reconfiguration of group communication systems. In R. Cleaveland, editor, 5^{th} *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 1579 in Lecture Notes in Computer Science, pages 104–118. Springer Verlag, 1999. 1

[Lam94] Leslie Lamport. Introduction to TLA. Technical Report 1994-001, DIGITAL SRC, Palo Alto, CA, 1994. 4.7

[Ler93] Xavier Leroy. *The Caml Light System, release 0.6: Documentation and User's Manual*, September 1993. 1.3

[LT89] Nancy Lynch and Mark Tuttle. An introduction to Input/Output automata. *Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands*, 2(3):219–246, September 1989. Also Tech. Memo MIT/LCS/TM-373. 4, 4.7

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996. 4, 4.7

[Men87] Paul Francis Mendler. *Inductive Definition in Type Theory*. PhD thesis, Cornell University, September 1987. 87–870. 11.1, 14.1

[Mey88] Bertrand Meyer. *Object–Oriented Software Construction*. Prentice–Hall, 1988. 10.6

[MHF93] John C. Mitchell, Furio Honsell, and Kathleen Fisher. A lambda calculus of objects and method specialization. In *IEEE Symposium on Logic in Computer Science*, 1993. 10.4.1, 10.6

[ML75] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In *Logic Colloquium '73*, pages 73–118. North–Holland, 1975. 11.1, 14, 14.3.2

[ML82] Per Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–75. North-Holland, Amsterdam, 1982. 14

[ML84a] Per Martin-Löf. *Intuitionistic Type Theory, Studies in Proof Theory, Lecture Notes*. Bibliopolis, Napoli, 1984. 14

[ML84b] Per Martin-Löf. *Intutionistic Type Theory*. Bibliopolis, Napoli, 1984. 14.3.2

[MN95] Olaf Müller and Tobias Nipkow. Combining model checking and deduction for I/O-automata. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019 of *LNCS*, pages 1–16. Springer, 1995. 1

[MOM93] Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, SRI International, 1993. 5.3

[MOM94] N. Martí-Oliet and J. Meseguer. General logics and logical frameworks. In D.M. Gabbay, editor, *What is a Logical System?* Oxford University Press, 1994. 5.3

[N.d91] N.deBruijn. A plea for weaker frameworks. In G. Huet and G. D.Plotkin, editors, *Logical Frameworks*. Cambridge University Press, 1991. 5.2

[Nei96] Gil Neiger. A new look at membership services. In *Proc.15thAnnual ACM Symposium on Principles of Dist. Comp.*, pages 331–340, May 1996. 4.7

[NGV94] R. P. Nederpelt, J. H. Geuvers, and R. C. De Vrijer, editors. *Selected Papers in Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, Amsterdam, 1994. 5.3

[NP92] Tobias Nipkow and Lawrence C. Paulson. Isabelle-91. In *Proceedings of the 11th International Conference on Automated Deduction*, pages 673–676, 1992. 1, 1.3, 5.2, 5.3, 7.5

[NvO98] Tobias Nipkow and David von Oheimb. Java-light is type-safe — definitely. In *Proc. 25th ACM Symp. Principles of Programming Languages*, pages p. 161–170. ACM Press, New York, 1998. 1

[Pau83] Laurence Paulson. A higher-order implementation of rewriting. *Sci. Computer Programming*, 3(2):119–149, 1983. 7.5

[Pau90] L.C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, London, 1990. 1.3

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer LNCS 828, 1994. 8.1, 9.1

[Pfe94] Frank Pfenning. Elf: A meta-language for deductive systems. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, pages 811–815. Springer LNAI 814, June 1994. 1

[Pit00] A. M. Pitts. Categorical logic. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 5. Algebraic and Logical Structures*, chapter 2. Oxford University Press, 2000. 10.6

[PM93] Christine Paulin-Mohring. Inductive Definitions in the systen Coq–Rules and Properties. In *International Conference on Typed Lambda Calculus and Applications*, pages 328–345, 1993. 10.3

[PT94] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994. 11.1, 11.5

[Rez85] Adrian Rezus. Semantics of constructive type theory. Technical Report 70, Nijmegen University, The Netherlands, September 1985. 11.1

[RT88] Robinson and Tennent. Bounded quantification and record-update problems. Message to `types` email list, 1988. 11.3

[RV97] Didier Rémy and Jérôme Vouillon. Objective ML: A simple object–oriented extension of ML. In *ACM Symposium on Principles of Programming Languages*, pages 40–53, 1997. 4

[Smi85] D.R. Smith. The design of divide and conquer algorithms. *Science of Computer Programming*, 5:37–58, 1985. 5.3

[Smi89] Scott Fraser Smith. *Partial Objects in Type Theory*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, NY, 1989. 11.3.3

[Smi90] Douglas R. Smith. KIDS: A semiautomatic program development system. *IEEE Trans. Softw. Eng.*, 16(9):1024–1043, September 1990. 5.3

[Smi91] D.R. Smith. KIDS—a knowledge-based software development system. In M.R. Lowry and R.D. McCartney, editors, *Automating Software Design*, pages 483–514. AAAI Press / MIT Press, Menlo Park, CA, 1991. 5.3

[SP92] G. Sutcliffe and J. Pinakis. A heterogeneous parallel deduction system. In *Proceedings of the Workshop on Automated Deduction, FGCS'92*, 1992. 9.7

[Sta83] The Nuprl Staff. PRL: Proof refinement logic programmer's manual (lambda prl, vax version). Cornell University, Dept. of Computer Science, 1983. 1

[Wan87] Mitchell Wand. Complete Type Inference for Simple Objects. In *Symposium on Logic in Computer Science*, pages 37–44. IEEE, June 1987. 10.6

[Wan89] Mitchell Wand. Type Innference for Record Concatenation and Multiple Inheritance. In *Fourth Annual Symposium on Logic in Computer Science*, pages 92–97. IEEE, June 1989. 10.6

[Zip] Richard Zippel. MathBus. Available online at `www.cs.cornell.edu` in the directory `Simlab/papers/mathbus/mathTerm.htm`. 7.9

Index

Conversions
- *addrC*, 308
- *andthenC*, 307
- *cutC*, 309
- *failC*, 307
- *firstC*, 307
- *foldC*, 309
- *higherC*, 308
- *idC*, 307
- *lowerC*, 308
- *orelseC*, 307
- *repeatC*, 307
- *rw*, 306
- *rwh*, 309
- *someSubC*, 308
- *sweepDnC*, 308
- *sweepUpC*, 309
- *tryC*, 307

Resources
- *auto_resource*, 312
- *decide_resource*, 493
- *reduce*, 310
- *squash_resource*, 408
- *sub_resource*, 426
- *toploop_resource*, 297

Rewrites
- *reduce_add*, 507
- *reduce_append_cons*, 550
- *reduce_append_nil*, 550
- *reduce_assoc_cons*, 551
- *reduce_assoc_nil*, 551
- *reduce_ball2_cons_cons*, 551
- *reduce_ball2_cons_nil*, 551
- *reduce_ball2_nil_cons*, 550
- *reduce_ball2_nil_nil*, 550
- *reduce_band_false*, 496
- *reduce_band_true*, 496
- *reduce_beta*, 445
- *reduce_bimplies_false*, 496
- *reduce_bimplies_true*, 496
- *reduce_bnot_false*, 496
- *reduce_bnot_true*, 496
- *reduce_bor_false*, 496
- *reduce_bor_true*, 496
- *reduce_bunion*, 625
- *reduce_dall*, 604
- *reduce_dexists*, 608
- *reduce_div*, 507
- *reduce_eq*, 507, 591
- *reduce_equal*, 591
- *reduce_fix*, 445
- *reduce_fold_left_cons*, 553
- *reduce_fold_left_nil*, 553
- *reduce_ifthenelse_false*, 496
- *reduce_ifthenelse_true*, 496
- *reduce_is_nil_cons*, 549
- *reduce_is_nil_nil*, 549
- *reduce_length_cons*, 554
- *reduce_length_nil*, 554
- *reduce_listindCons*, 542
- *reduce_listindNil*, 542
- *reduce_lt*, 507
- *reduce_map_cons*, 553
- *reduce_map_nil*, 553
- *reduce_mem*, 598
- *reduce_mul*, 507
- *reduce_nth_cons*, 554
- *reduce_rem*, 507
- *reduce_replace_nth_cons*, 555
- *reduce_rev_assoc_cons*, 552
- *reduce_rev_assoc_nil*, 552

reduce_rev_cons, 555
reduce_rev_nil, 555
reduce_scoll, 640
reduce_sep, 618
reduce_set_ind, 586
reduce_sub, 507
reduce_tree_eta, 528
reduce_tree_ind, 527
reduceDecideInl, 438
reduceDecideInr, 439
reduceFst, 467
reducePrecind, 532
reduceSnd, 467
reduceSpread, 467
rewriteAxiom2, 317
unfold_all, 477
unfold_and, 476
unfold_append, 550
unfold_assert, 496
unfold_assoc, 551
unfold_ball2, 550
unfold_band, 495
unfold_bfalse, 495
unfold_bimplies, 495
unfold_bisect, 518
unfold_bnot, 496
unfold_bool, 495
unfold_bor, 495
unfold_btrue, 495
unfold_bunion, 524, 625
unfold_cand, 476
unfold_collect, 585
unfold_cor, 476
unfold_dall, 604
unfold_decidable, 492
unfold_dexists, 607
unfold_dfun, 454
unfold_dfun_prop, 592
unfold_dfun_prop2, 592
unfold_empty, 621
unfold_eq, 591
unfold_equal, 591
unfold_esquash, 569
unfold_esquash_bool, 569
unfold_exists, 477
unfold_false, 476
unfold_fold_left, 553
unfold_fun, 458
unfold_fun_prop, 592
unfold_fun_set, 591
unfold_ge, 507
unfold_gt, 507
unfold_iff, 476
unfold_ifthenelse, 495
unfold_implies, 476
unfold_inf, 633
unfold_is_nil, 549
unfold_isset, 585
unfold_le, 507
unfold_length, 554
unfold_lt, 633
unfold_map, 553
unfold_mem, 598
unfold_not, 476
unfold_nth, 554
unfold_or, 476
unfold_pair, 630
unfold_partial_order, 462
unfold_prod, 472
unfold_prop, 476
unfold_rel, 638
unfold_replace_nth, 555
unfold_restricted, 618
unfold_rev, 555
unfold_rev_assoc, 552
unfold_sall, 611
unfold_scoll, 640
unfold_sep, 618
unfold_set, 585
unfold_set_ind, 585
unfold_sexists, 613
unfold_sing, 622
unfold_srecind, 537
unfold_subset, 615
unfold_succ, 633
unfold_top, 513

unfold_true, 476
unfold_union, 626
unfold_well_founded, 463
unfold_zero, 633
unfoldFst, 467
unfoldSnd, 467
Rules
all_elim, 486
all_intro, 485
all_type, 485
all_univ, 485
and_elim, 479
and_intro, 479
and_type, 479
and_univ, 479
append_wf2, 556
applyEquality, 457
assert_band_elim, 503
assert_band_intro, 504
assert_bimplies_elim, 503
assert_bimplies_intro, 504
assert_bnot_elim, 502
assert_bnot_intro, 502
assert_bor_elim, 502
assert_bor_intro_left, 503
assert_bor_intro_right, 503
assert_decidable, 492
assert_false, 500
assert_magic, 502
assert_true, 500
assert_type, 500
assertSquashElim, 504
assoc_wf, 556
atomEquality, 432
atomSqequal, 433
atomType, 432
axiomMember, 413
ball2_wf2, 556
band_member, 500
bfalse_member, 497
bimplies_member, 500
bisectEliminationLeft, 519
bisectEliminationRight, 519
bisectEquality, 518
bisectMemberEquality, 519
bisectSubtypeBelow, 520
bisectSubtypeLeft, 519
bisectSubtypeRight, 519
bisectType, 518
bnot_member, 500
bool_ext_equality, 501
bool_subst_concl, 501
bool_subst_hyp, 501
boolContradiction1, 498
boolContradiction2, 498
boolElimination2, 497
boolEquality, 497
boolSqequal, 499
boolType, 497
bor_member, 499
btrue_member, 497
bunion_fun, 628
bunion_isset, 626
bunion_member_elim, 628
bunion_member_intro_left, 627
bunion_member_intro_right, 627
bunionElimination, 525
bunionEquality, 524
bunionMemberEqualityLeft, 525
bunionMemberEqualityRight, 525
bunionType, 524
cand_elim, 483
cand_intro, 483
cand_type, 483
cand_univ, 483
collection, 643
cons_neq_nil, 547
consEquality, 544
cor_elim, 485
cor_intro_left, 484
cor_intro_right, 484
cor_type, 484

cor_univ, 484
cut, 417
dall_elim, 605
dall_fun, 606
dall_intro, 605
dall_res, 606
dall_type, 605
dec_false, 493
dec_true, 493
decideEquality, 441
dexists_elim, 609
dexists_fun, 609
dexists_intro, 608
dexists_res2, 610
dexists_type, 608
elem_isset, 599
empty_isset, 622
empty_member_elim, 622
eq_concl_subst, 596
eq_equality1, 592
eq_equality2, 593
eq_fun, 595
eq_hyp_subst, 595
eq_isset, 595
eq_ref, 594
eq_restricted, 618
eq_sym, 594
eq_trans, 594
eq_type, 593
equal_fun, 595
equal_intro, 593
equal_isset_left, 593
equal_isset_right, 594
equal_set, 586
equal_type, 593
equality_squashElimination, 414
equalityAxiom, 410
equalityElimination, 413
equalityEquality, 412
equalityFormation, 411
equalityRef, 411
equalitySym, 411
equalityTrans, 411
equalityType, 412
equalityTypeIsType, 412
esquash_bool_false, 570
esquash_bool_true, 570
esquash_bool_type, 570
esquash_bool_univ, 570
esquash_equal_elim, 571
esquash_equal_elim2, 572
esquash_equal_intro, 572
esquash_esquash_elim, 571
esquash_esquash_elim2, 572
esquash_intro, 571
esquash_type, 570
esquash_unit_elim2, 572
esquash_univ, 570
esquash_void_elim, 571
esquash_void_elim2, 572
exists_elim, 487
exists_intro, 486
exists_type, 486
exists_univ, 486
false_elim, 478
false_member, 477
false_squash, 478
false_type, 478
false_univ, 477
fold_left_wf, 557
fun_prop, 597
fun_prop_type, 596
fun_ref, 597
fun_set, 596
fun_set_type, 596
functionElimination, 456
functionEquality, 454
functionExtensionality, 456
functionSubtype, 457
functionType, 455
hypothesis, 416
hypReplacement, 419
hypSubstitution, 419
iff_elim, 482
iff_intro, 482

iff_type, 482
iff_univ, 482
ifthenelse_equality, 499
ifthenelse_type2, 498
implies_elim, 482
implies_intro, 481
implies_type, 481
implies_univ, 481
independentApplyEquality, 461
independentFunctionElimination, 460
independentFunctionElimination2, 460
independentFunctionEquality, 458
independentFunctionType, 458
independentLambdaEquality, 459
independentLambdaFormation, 459
independentPairEquality, 473
independentPairFormation, 473
independentProductElimination, 473
independentProductEquality, 472
independentProductFormation, 472
independentProductSubtype, 474
independentProductType, 472
indEquality, 509
inf_isset, 634
inlEquality, 440
inlFormation, 440
inrEquality, 441
inrFormation, 440
intElimination, 508
intEquality, 508
intersectionElimination, 515
intersectionEquality, 513
intersectionMemberCaseEquality, 515
intersectionMemberEquality, 514
intersectionSubtype, 516
intersectionSubtype2, 516
intersectionSubtype3, 516
intersectionType, 513
introduction, 418
intType, 508
is_nil_wf, 556
isset_assum, 586
isset_collect, 587
isset_collect2, 587
lambdaEquality, 455
lambdaFormation, 455
length_wf, 558
less_than_memberEquality, 511
less_thanEquality, 511
list_indEquality, 545
listElimination, 544
listEquality, 543
listSubtype, 546
listType, 543
lt_fun, 635
lt_restricted, 635
map_wf, 557
mem_equal, 599
mem_fun_left, 600
mem_type, 599
member_fun, 600
member_intro, 599
member_restricted, 618
nat_elim, 635
nil_neq_cons, 546
nilEquality, 543
nilFormation, 543
nilSqequal, 546
not_elim, 479
not_intro, 478

not_type, 478
not_univ, 478
nth_wf, 558
numberEquality, 508
or_elim, 481
or_intro_left, 480
or_intro_right, 480
or_type, 480
or_univ, 480
pair_fun, 631
pair_isset, 630
pair_member_elim, 631
pair_member_intro_left, 630
pair_member_intro_right, 630
pairEquality, 469
pairFormation, 468
partial_order_type, 463
power_isset1, 640
power_thm, 641
precElimination, 534
precEquality, 533
precindEquality, 535
precMemberEquality, 533
precUnrollElimination, 535
productElimination, 469
productEquality, 468
productSubtype, 471
productType, 468
quotient_equalityElimination, 566
quotient_memberEquality, 565
quotient_memberWeakEquality, 564
quotientElimination1, 565
quotientElimination2, 566
quotientEquality, 563
quotientMember, 564
quotientSubtype, 567
quotientType, 562
quotientWeakEquality, 563
rel_type, 638
replace_nth_wf, 558
rev_assoc_wf, 557
rev_wf, 558
rewrite_squashElimination, 414
rewriteAxiom1, 317
rewriteSym, 318
rfunction_applyEquality, 452
rfunction_lambdaEquality, 450
rfunction_lambdaFormation, 449
rfunction_rfunction_subtype, 452
rfunctionElimination, 451
rfunctionEquality, 448
rfunctionExtensionality, 451
rfunctionType, 449
sall_elim, 612
sall_intro, 611
sall_type, 611
sep_elim, 620
sep_fun, 620
sep_intro2, 619
sep_isset, 619
set_elim, 573, 587
set_ext, 600
set_ind_equality2, 588
set_induction, 643
set_isset, 599
set_subtype, 436
set_type, 586
setElimination, 436
setEquality, 434
setMemberEquality, 435
setMemberFormation, 435
setType, 434
sexists_elim, 614
sexists_intro, 613
sexists_type, 613
sing_fun, 623
sing_isset, 623
sing_member_elim, 623

sing_member_intro, 623
spreadEquality, 470
squashFromAny, 414
srec_memberEquality, 538
srecElimination, 539
srecEquality, 538
srecindEquality, 540
srecType, 538
srecUnrollElimination, 539
subset_collection, 644
subset_elim, 616
subset_fun, 616
subset_intro, 615
subset_res, 616
subset_type, 615
substitution, 419
subtype_axiomEquality, 425
subtype_axiomFormation, 424
subtype_squashElimination, 426
subtypeElimination, 425
subtypeElimination2, 425
subtypeEquality, 423
subtypeType, 423
subtypeTypeLeft, 424
subtypeTypeRight, 424
succ_fun, 634
succ_isnat, 634
succ_isset, 634
succ_member_elim1, 637
succ_member_intro1, 636
thin, 417
tokenEquality, 432
tokenFormation, 432
topMemberEquality, 515
topSubtype, 517
topType, 514
topUniv, 514
tree_indEquality, 530
treeEquality, 529
true_intro, 477
true_member, 477
true_type, 477
true_univ, 477
tunionElimination, 523
tunionEquality, 521
tunionMemberEquality, 522
tunionMemberFormation, 522
tunionType, 521
type_squashElimination, 414
typeEquality, 413
unhideEqual, 437
unhideGoalEqual, 437
union_fun, 629
union_isset, 626
union_member_elim, 628
union_member_intro, 627
unionElimination, 441
unionEquality, 439
unionSubtype, 442
unionType, 439
unit_memberEquality, 429
unit_memberFormation, 429
unit_squashElimination, 430
unitElimination, 430
unitEquality, 429
unitSqequal, 431
unitType, 429
universeAssumType, 415
universeCumulativity, 415
universeMember, 414
universeMemberType, 415
universeType, 415
void_squashElimination, 428
void_subtype, 428
void_well_founded, 454
voidElimination, 427
voidEquality, 427
voidType, 427
wElimination, 529
well_founded, 447
well_founded_apply_type, 447
well_founded_assum_elim, 446

well_founded_reduction, 464
well_founded_type, 463
wEquality, 528
wType, 528
zero_isnat, 634
zero_isset, 634
zero_member_elim, 636
zero_member_intro, 636

Tactics
addHiddenLabelT, 304
assertAtT, 421
assertT, 421
assumT, 489
backThruAssumT, 490
backThruHypT, 488
completeT, 301
cutT, 299
decideT, 493
eqSetRefT, 597
eqSetSymT, 597
eqSetTransT, 597
esquashT, 573
failT, 299
failwithT, 299
firstT, 300
genAssumT, 490
genUnivCDT, 488
idT, 299
ifLabT, 305
instHypT, 488
keepingLabelT, 304
memberOfT, 601
moveToConclT, 487
moveToConclVarsT, 487
natIndT, 637
nthAssumT, 299
nthHypT, 420
onAllClausesT, 304
onAllHypsT, 304
onClausesT, 303
onClauseT, 303
onConclT, 303
onHypsT, 303
onHypT, 303
onSomeHypT, 304
onVarT, 303
orelseT, 300
powerT, 641
progressT, 302
removeHiddenLabelT, 304
repeatForT, 302
repeatMT, 305
repeatT, 302
selT, 302
seqOnSameConclT, 300
seqT, 300
setExtT, 601
setOfT, 601
setSubstT, 597
splitBoolT, 504
splitITE, 504
squashT, 409
substT, 422
then_OnFirstT, 301
then_OnLastT, 301
then_OnSameConclT, 301
thenAT, 305
thenMT, 305
thenT, 300
thenWT, 305
thinAllT, 420
thinT, 420
timingT, 301
tryT, 301
univCDT, 488
useWitnessT, 421
withBoolT, 302
withIntT, 302
withT, 302
withTypeT, 302

Terms
add, 506
all, 475
and, 475
append, 547

apply, 444
assert, 495
assoc, 548
atom, 431
axiom, 293
ball2, 548
band, 495
bfalse, 494
bimplies, 495
bind, 416
bisect, 517
bnot, 495
bool, 494
bor, 494
btrue, 494
bunion, 524
cand, 475
collect, 585
comment, 291
cond_rewrite, 292
cons, 542
cor, 475
cumulativity, 410
dall, 604
decidable, 492
decide, 438
dexists, 607
dform, 294
dfun_prop, 590
div, 506
empty, 621
eq, 590
equal, 410, 590
esquash, 569
esquash_bool, 569
except_mode_df, 294
exists, 475
false, 475
fix, 444
fold_left, 548
fst, 467
fun, 444
fun_prop, 590
fun_set, 590
ge, 506
goal, 295
gt, 506
hide, 434
iff, 475
ifthenelse, 495
implementation, 291
implies, 475
ind, 506
inf, 633
infix, 293
inherit_df, 294
inl, 438
inr, 438
int, 506
interface, 291
is_nil, 547
isect, 513
isset, 585
lambda, 444
le, 506
length, 549
list, 542
list_ind, 542
lt, 506, 633
map, 548
mem, 598
member, 598
meta_function, 294
meta_iff, 294
meta_implies, 294
meta_labeled, 294
meta_theorem, 294
meta_theory, 294
mode_df, 294
mul, 506
nil, 542
not, 475
nth, 549
number, 506
opname, 292
or, 475

pair, 467, 629
parens_df, 294
parent, 292
partial_order, 462
power, 639
prec, 294, 532
prec_df, 294
prec_rel, 294
precind, 532
prod, 467
prop, 475
quot, 562
rel, 638
rem, 506
replace_nth, 549
resource, 293
resource_defs, 293
restricted, 617
rev, 549
rev_assoc, 548
rewrite, 292
rfun, 444
rule, 293
sall, 611
sep, 617
set, 434, 585
set_ind, 585
sexists, 612
sing, 622
snd, 467
spread, 467
srec, 537
srecind, 537
status, 295
status_asserted, 295
status_bad, 295
status_complete, 295
status_partial, 295
sub, 506
subset, 614
subtype, 423
succ, 633
summary_item, 293
token, 431
tree, 527
tree_ind, 527
true, 475
tunion, 521
type, 410
union, 438, 625
univ, 410
w, 527
well_founded, 444, 462
well_founded_apply, 444
well_founded_assum, 444
well_founded_prop, 444
zero, 633