Information-Intensive Proof Technology

Marktoberdorf NATO Summer School 2003 *

Robert L. Constable
Cornell University

*This work was supported by the DoD Multidisciplinary University Research Initiative (MURI) program administered by the Office of Naval Research under Grant N00014-01-1-0765, and by the National Science Foundation under Grant CCR-0208536.

Contents

Chapter 1

Series Introduction and Type Theory Basics

1.1 Introduction

We are going to stress the theme that theorem provers can accomplish more if they work together. It seems clear that there will always be several different provers supported at any one time. They will use different logics, incompatible logics, and yet we will want to collect their general results together and attempt to share them and use them in research and applications. How can this be done?

There are groups collecting formal mathematics and presenting it on the Web, e.g. HELM [15], OMDoc [70], and our Formal Digital Library [9]. Our project is concerned with the logical issues behind organizing these collections, and with some of the digital library issues, such as collecting formal metadata.

The plan of these lectures is to first explore the logical problems of sharing and show some solutions, referring to the work of Doug Howe [67, 66], Pavel Naumov [92], Messeguer and Stehr [86], and recent work of Evan Moran [90]. Then we will look at some technical issues of a logical nature in creating a logical library of formalized mathematics.

In the last two lectures I will illustrate how provers with sufficiently large libraries are able to formalize and verify protocols at speeds close to those of the designers and programmers as they create them. This will also be an opportunity to discuss formal elements of our approach to formalizing concepts from distributed computing [23].

1.2 Type Theory Basics

Consider these provers: ACL2, Alf, Coq, HOL, Isabelle, MetaPRL, Minlog, Mizar, Nuprl, PVS, and Twelf [102]. They are all interactive. Nine of the eleven are based on higher-order typed logic. ACL2 is first-order and Mizar is based on set theory; it is called Banach/Tarski, like ZFC+ Inaccessible Cardinals. Why are type theory and typed logic (higher-order logic) so dominant?

1. Types connect well to programming.
2. Types capture a level of abstraction at which mathematicians write, a layer on top of set theory.
3. Type theory is capable of expressing all of classical and computational mathematics. There is potential to also capture concurrent computational mathematics as a layer on top of type theory [23]

Let us examine the elements of three representative typed logics:

HOL	—	Based on Church's Simple Theory of Types [51], an expressive but conservative foundation, used in hardware and software verification [51]
Isabelle-HOL	—	Closely related to HOL [99]
PVS	—	An extension of simple type theory that has achieved considerable popularity and success, heavily used in software verification [98, 97]
Nuprl	—	One of the first constructive type theories, related to Martin-Löf's type theories, heavily used in software verification [40, 39]

1.3 Common Elements of the Logical Language

All three theories are based on a rich type system that includes:

- Atomic types: Booleans ($\mathbb{B}$), natural numbers ($\mathbb{N}$)
- Compound types:
  - $A \rightarrow B$ function space (total functions) — elements are lambda terms, $\lambda x.b$
  - $A \times B$ Cartesian products — elements are ordered pairs, $< a, b >$
  - $\mu X.F$ inductive (recursive types)

They all support *higher order logic* which quantifies over functions and predicates. Essentially, the propositions are built from:

- Atomic propositions: $True, False, a = b$ in A
- Propositional connectives: &, $v, \Rightarrow$
- Typed quantifiers: $\forall x : A, \exists x : A$ (HOL uses type variables, $\forall x_A, \exists x_A$)

But several details are very different. Here are six salient contrasts:

1. HOL and PVS have a standard set theoretic semantics, whereas Nuprl's standard semantics is based on partial equivalence relations ("pers") over algorithms and data constructors. Its functions are *computable* and *polymorphic* (applying to many types, e.g. $\lambda x.x$).

2. The HOL and PVS logics are *classical.* They support the law of excluded middle, $P \vee \neg P$ (where $\neg P$ is $P \Rightarrow False$). The Nuprl logic is defined from the type theory based on the *propositions-as-types* principle, and is thus *constructive.*

3. Relations on a type A in HOL and PVS are functions from A into the Booleans, e.g. $A \rightarrow \mathbb{B}$. In Nuprl, there is hierarchy of types of propositions, $Prop_i$, and relations are propositional functions $A \rightarrow Prop_i$ (thus Nuprl is *predicative* while HOL and PVS are *impredicative*).

4. Nuprl and PVS use *dependent types* for:

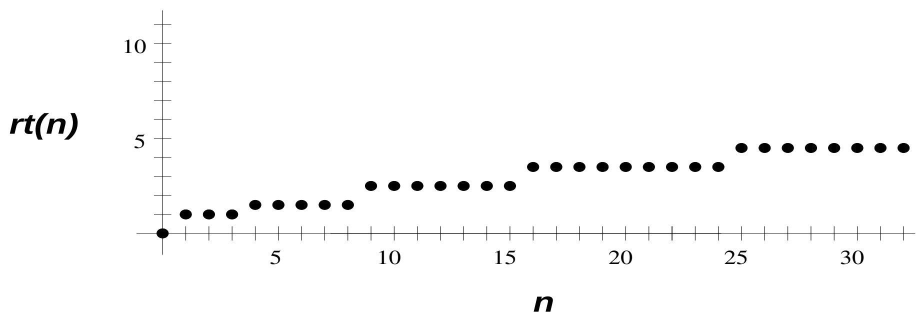
   - functions $x : A \rightarrow B(x)$; elements are $\lambda x.b$.
   - products $x : A \times B(x)$; elements are pairs, $< a, b >$.
   - records $\{x_1 : A_1; x_2 : A_2(x); \ldots ; x_n : A_n(x_1, \ldots , x_{n-1})\}$; elements are functions from identifiers to elements of the A_i.

5. All types in HOL are non-empty whereas Nuprl and PVS allow *empty types.*
6. HOL and PVS adopt the *Axiom of Choice* (HOL uses Hilbert's choice operator, $\epsilon\ x.P$). In Nuprl the axiom of choice is a consequence of the *propositions-as-types principle.*
7. Nuprl types are also objects that belong to other types, creating a hierarchy of "large types" (called *universes*, denoted $\mathbb{U}_i$). In HOL and PVS, types are not objects. So in a sense Coq and Nuprl are *type theories,* whereas the others are *typed logics.**
8. Nuprl supports *partial functions* and a *domain theory* based on the notion that the underlying computation system is the untyped λ-calculus with constants.

1.4 Type Theory as a Specification Language

The logical language of type theory is very expressive and interesting as a *specification language,* for both mathematics and computer science. Here are some simple examples of how we can specify computational problems.

Integer Square Roots Suppose we want to specify the integer square root of a natural number, e.g. $rt(0) = 0, rt(1) = 1, rt(2) = 1, rt(3) = 1, rt(4) = 2, \ldots, rt(8) = 2, rt(9) = 3$.



Can we write the specification as $rt(n)^2 = n$? We want $rt : \mathbb{N} \to \mathbb{N}$, so the type would be wrong. We would need to produce *real numbers* as values, so if we denote reals by $\mathbb{R}$, then the type would be $rt : \mathbb{N} \to \mathbb{R}$, not even $rt : \mathbb{N} \to \mathbb{Q}$, since $rt(n)$ is irrational unless n is a square. Writing $rt(n^2) = n$ is not a complete specification.

What we demand of root is that $rt(n)^2 \le n$. We would get this if we took $\sqrt{\ }\ :\ \mathbb{N} \to \mathbb{R}$ and defined $rt(n) = \lfloor\sqrt{n}\rfloor$, since $\lfloor\sqrt{n}\rfloor \le \sqrt{n}$ and $\lfloor\sqrt{n}\rfloor^2 \le n$.

Defining $\sqrt{n}$ is a more difficult task. We would want $\sqrt{\ }\ :\ \mathbb{R} \to \mathbb{R}$ and we'd need the whole apparatus of $\mathbb{R}$. In all three systems — HOL, Nuprl, and PVS — we could do this, but in Nuprl the reals would be computable numbers, and the mapping $\lfloor\ \rfloor : \mathbb{R} \to \mathbb{N}$ is not computable, so it does not exist as a function. Indeed, there is no nontrivial computable function from $\mathbb{R} \to \mathbb{N}$. See Bridges and Bishop [31].

So, we want to define $rt : \mathbb{N} \to \mathbb{N}$ and we know $rt(n)^2 \le n$. However, this spec would allow $rt(n) = 0$ for all n. We want the largest number r such that $r^2 \le n$; i.e., we need that $(r+1)^2 > n$. So in the end we want the least r such that $r^2 \le n\ \&\ n < (r+1)^2$. Let us call this relation $Root(n, r)$. This specifies r uniquely. So a good specification is

Thm 1.1 $\forall n : \mathbb{N}.\ \exists r : \mathbb{N}.\ r^2 \le n < (r+1)^2$

*In Nuprl, the primitive type equality is *intensional,* e.g. $A_1 \to B_1 = A_2 \to B_2$ iff $A_1 = A_2, B_1 = B_2$. In HOL and PVS the type checkers assume an intensional equality as well.

Should we say instead

Thm 1.2 $\forall n : \mathbb{N}.\ \exists! r : \mathbb{N}.\ Root(n, r)$

because then we could use the *implicit function theorem* to show this, valid in all three logics.

Cor 1.1 $\exists rt : \mathbb{N} \to \mathbb{N}.\ \forall n : \mathbb{N}.\ Root(n, rt(n))$

In Nuprl we also obtain from Thm 1.1 an object called an *extract:*

Cor 1.2 $ext(\textit{Thm 1.1}) \epsilon\ \mathbb{N} \to \mathbb{N}$

Cor 1.3 $\forall n : \mathbb{N}.\ Root(n, ext(\textit{Thm 1.1})(n))$

Nuprl proofs of Theorem 1.1 are given in Appendix A. To impart the syntactic flavor of HOL and PVS, let's write the specification in them. In PVS and Nuprl, $\mathbb{N}$ must be defined.

Nuprl	$\mathbb{N} == \{i : \mathbb{Z} \mid 0 \leq i\}$
PVS	nat : NONEMPTY_TYPE $= \{x : real \mid is\_nat(x)\}$

Thm 1–HOL	$!\, x.?\, r.\, Root\ n, r$
Thm 1–Nuprl	$\forall n : \mathbb{N}.\ \exists r : \mathbb{N}.\ Root(n, r)$
Thm 1–PVS	(FORALL $[n : nat]$: (EXISTS $[r : nat]$: $Root(n, r)$))

The precise meaning of these expressions depends on the mathematical semantics of the languages. This is usually given informally, in ordinary careful logic. The semantics of HOL is the most standard and straight forward, written by Andy Pits [51], in Zermelo set theory.

The semantics for PVS is based on ZFC set theory, but it also depends on its proof theory, because the notion of a well-formed PVS expression is not recursive — not syntactic. Thus proof theory and semantics are intermixed in a delicate way.

The semantics for Nuprl is based on Martin-Löf's semantic methods, to account for the difficulty that well-formedness of expressions is not recursive. This semantic method is profoundly novel and powerful. It could apply to PVS as well, but that has not been done. Therein lies a PhD thesis.

Let's review the semantic ideas briefly. In HOL the natural numbers are an inductively defined subset of *infinite,* a postulated infinite set. A relation such as $Root$ is a total function from num to $(num \to bool)$. A function is a single-valued relation, and a relation is a set of ordered pairs. The universal quantifier is a logical operator defined from the existential quantifier, which is in turn defined by the choice operator.

$$\begin{array}{rcl} x!.Rx & = & \neg ?x.\neg Rx \\ \neg R & = & R \Rightarrow False \\ False & = & 0 = 1 \\ ?x.R & = & (R\epsilon(R)) \end{array}$$

The choice operator ϵ takes a function from $A \to bool$, and returns an element of $\{x : A \mid Rx = true\}$, if there is one, otherwise an element of A.

Before this expression is given meaning, it must be checked for syntactic correctness. A type checker will infer the type of x, r from the type given for R in its definition, in this case $num \to (num \to bool)$.

The PVS semantics for Thm 1.1 is similar to that for HOL, except that subtyping is explicit, and the logical operators are defined differently. For example,

$$\text{FORALL } [x : A] : P(x) == \text{LAMBDA } [x : A] : P(x) = \text{LAMBDA } [x : A] : True$$

The well-formedness of Theorem 1.1 is a matter of proof, not semantics. It must be shown that $\{x : real \mid is\_nat(x)\}$ is a type, and that $Root$ is defined over it.

In the case of Nuprl, the semantics defines what it means for an expression to be a proposition (type), and it must be *true* that the term $\forall n : \mathbb{N}.\ \exists r : \mathbb{N}.\ Root(n, r)$ is a proposition. Knowing this requires knowing that $\mathbb{N}$ is a type and that $Root$ is a function $\mathbb{N} \to (\mathbb{N} \to Prop)$. Unlike for PVS, these are first matters for the semantics, then for the proof system.

We follow Martin-Löf's semantic method. To know that $\mathbb{N}$ is a type, we must have a canonical name for the type; we must know what the *canonical elements* are, and when they are equal. Generally, an element belongs to $\mathbb{N}$ iff it reduces to a canonical element.

In the case of both HOL and Nuprl, the semantics accounts for the truth not only of judgments, but also of objects called *sequents,* or *hypothetical judgment.* These have the form $\overline{H} \vdash G$, where $\overline{H} = H_1, \ldots, H_n$ (possibly $n = 0$), and the H_i and G are judgments; the H_i are the *hypotheses,* and G the *conclusion* or *goal.*

The base form of a judgment is this:

HOL	$\vdash G$	judges that proposition G is true
Nuprl	$\vdash G\ ext\ g$	judges that g is evidence for G (or that type G is inhabited by g)

(Martin-Löf writes the judgment as $\vdash g\epsilon G$, and g is always explicit; Nuprl can suppress the element g in its display of judgments, so that they resemble historically earlier judgments, which are more familiar.)

In HOL the hypothetical judgment abbreviated $\overline{H} \vdash G$ is

$$H_1, \ldots, H_n \vdash G.$$

In Nuprl, the judgment includes indication of the evidence:

$$x_1 : H_1, \ldots, x_n : H_n \vdash G\ ext\ g.$$

The hypotheses can explicitly mention the previous x_i, so H_{i+1} can depend on $x_1, \ldots, x_i$, and G and g can depend on $x_1, \ldots, x_n$.

In HOL we define what it means for a model M to satisfy a sequent, written

$$\overline{H} \models_M G.$$

This is the standard *Tarski-style semantics.* The notion of satisfiability is used to define the concept of a *sound* proof rule in the deduction system.

In *Martin-Löf-style semantics,* the meaning of a judgment is given directly in reference to the terms occurring in it, and the computation rules on these terms are given as a structured operational semantics. (These are sometimes called *computational term models.*) The basic sequent

$$\vdash G\ ext\ g$$

means that G is a type and g is a member of it. The hypothetical judgment means that if H is a type and x a member of it, then $G[a/x]$ reduces to a type for each canonical $a\epsilon H$ and $g[a/x]$ reduces to a canonical member — where $g[a/x]$ denotes the substitution of a for each free occurrence of x in term g. Moreover, the sequence means that if $a_1 = a_2$ in H, then $G[a_1/x] = G[a_2/x]$ and $g[a_1/x] = g[a_2/x]$ in G. This property is called *functionality,* and it is fundamental.

In his influential PhD Thesis [11], Stuart Allen gave a non-type theoretic mathematical account of Martin-Löf's semantic method. He defines the notion of a type system and the concept of a functional sequent. A rule of inference is *sound* in a type system if the conclusion is functional when the hypotheses are. Allen shows that Martin-Löf's 1982 theory [84] and Nuprl [40] both have sound rules. We will examine some of these rules as we describe the deduction systems.

Note, PVS does not provide a Tarski semantics, because its notion of well-formed proposition is not decidable. It also does not use a term model semantics. Instead, Owre and Shankar [98] use a "hybrid semantics," which depends on the deduction system and thus cannot be used to show the soundness of the rules. Some of their ideas depend on the work of Howe [67] and Dybjer [45], which suggests that those methods might offer a means of establishing soundness.

1.5 Counting Library Example

by Stuart Allen

This entire section is material taken from Stuart Allen's Web Booklet, *Discrete Math Materials,* in the Nuprl Mathematics Library at http://www.nuprl.org/Nuprl4.2/Libraries/

1.5.1 Counting is finding a function of a certain kind.

When we count a class of objects, we generate an enumeration of them, which we may represent by a 1-1-CORRESPONDENCE from a standard class having that many objects to the class being counted. Our standard class of n objects, for $n \in \mathbb{N}$, will be $\mathbb{N}_n$, the class $\{k{:}\mathbb{Z} \mid 0 \leq k < n\}$ of natural numbers less than n. A more familiar choice of standard finite classes might have been $\{k{:}\mathbb{Z} \mid 1 \leq k \leq n\}$, but there is also another tradition in math for using $\{k{:}\mathbb{Z} \mid 0 \leq k < n\}$.

So, a class A has n members just when

$\exists f{:}(\mathbb{N}_n {\to} A)$. Bij($\mathbb{N}_n$; A; f)

which may also be expressed as

($\mathbb{N}_n \sim A$)

since ($\exists f{:}(A{\to}B)$. Bij(A; B; f)) $\Leftrightarrow$ ($A \sim B$), or as

($A \sim \mathbb{N}_n$) since ($A \sim B$) $\Rightarrow$ ($B \sim A$).

Now, since counting means coming up with an enumeration, we may ask whether counting in different ways, i.e., coming up with different orders, will always result in the same number, as we assume. Of course, we know this is so, but there are different degrees of knowing. It is not necessary to simply accept this as an axiom; there is enough structure to the problem to make a non-trivial proof.

Thm* $(A \sim \mathbb{N}_m) \Rightarrow (A \sim \mathbb{N}_k) \Rightarrow m = k$

This theorem is closely related to what is sometimes called the "pigeon hole principle," which states the mathematical content of the fact that if you put some number objects into fewer pigeon holes, then there must be at least two objects going into the same pigeon hole. Number the pigeon holes with the members of $\mathbb{N}_k$, and the objects with the members of $\mathbb{N}_m$; then a way of putting the objects into the holes is a function in $\mathbb{N}_m {\to} \mathbb{N}_k$:

Thm* $\forall m,k{:}\mathbb{N}$, $f{:}(\mathbb{N}_m {\to} \mathbb{N}_k)$. $k{<}m \Rightarrow (\exists x,y{:}\mathbb{N}_m.\ x \neq y$ & $f(x) = f(y))$

If you examine the proofs of these theorems, you will notice that they both cite the key lemma

Thm* ($\exists f{:}(\mathbb{N}_m {\to} \mathbb{N}_k)$. Inj($\mathbb{N}_m$; $\mathbb{N}_k$; f)) $\Rightarrow m {\leq} k$.

1.5.2 Gloss of a formal proof

The codomain size of a finite injection is bounded by the domain size.

Thm* $(\exists f{:}(\mathbb{N}_m \rightarrow \mathbb{N}_k).\ \text{Inj}(\mathbb{N}_m;\ \mathbb{N}_k;\ f)) \Rightarrow m{\le}k$

where

Inj$(A;\ B;\ f)\ \equiv_{\text{def}}\ \forall a_1{,}a_2{:}A.\ f(a_1) = f(a_2) \in B \Rightarrow a_1 = a_2$

This will be proved using induction on m, varying k. The base case, $0{\le}k$, is trivial, so we move on to the induction step, assuming $0{<}m$, and assuming the induction hypothesis:

$\forall k'{:}\mathbb{N}.\ (\exists f'{:}(\mathbb{N}_{m-1} \rightarrow \mathbb{N}_{k'}).\ \text{Inj}(\mathbb{N}_{m-1};\ \mathbb{N}_{k'};\ f')) \Rightarrow m{-}1{\le}k'$.

The problem is then to show that $m{\le}k$, given some $f \in \mathbb{N}_m \rightarrow \mathbb{N}_k$ such that Inj$(\mathbb{N}_m;\ \mathbb{N}_k;\ f)$.

Obviously, $m{\le}k$ will follow from $m{-}1{\le}k{-}1$, so by applying the induction hyp to $k{-}1$, our problem reduces to finding an $f' \in \mathbb{N}_{m-1} \rightarrow \mathbb{N}_{k-1}$ such that Inj$(\mathbb{N}_{m-1};\ \mathbb{N}_{k-1};\ f')$.

Such a construction is

Replace k by $f(m)$ in $f\ \equiv_{\text{def}}$ Replace x s.t. $x{=}_2k$ by $f(m)$ in f

(Replace x s.t. $P(x)$ by y in $f)(i)\ \equiv_{\text{def}}$ if $P(f(i)){\rightarrow}\ y$ else $f(i)$ fi

Thm* Inj$(\mathbb{N}_{m+1};\ \mathbb{N}_{k+1};\ f) \Rightarrow$ Inj$(\mathbb{N}_m;\ \mathbb{N}_k$; Replace k by $f(m)$ in $f)$

This last theorem is sufficient for concluding our argument.

(Note: Considering $f \in \mathbb{N}_{k+1} \rightarrow \mathbb{N}_{j+1}$ as a sequence of k+1 values selected from the first j+1 natural numbers, (Replace j by $f(k)$ in $f) \in \mathbb{N}_k \rightarrow \mathbb{N}_j$ removes the entry for the largest value, namely j, and replaces it with the last value of the sequence, namely $f(k)$, if necessary.) QED

This is the key lemma to the proofs of the uniqueness of counting, and the pigeon hole principle, i.e.,

Thm* $(A \sim \mathbb{N}_m) \Rightarrow (A \sim \mathbb{N}_k) \Rightarrow m = k$, and

Thm* $\forall m{,}k{:}\mathbb{N},\ f{:}(\mathbb{N}_m \rightarrow \mathbb{N}_k).\ k{<}m \Rightarrow (\exists x{,}y{:}\mathbb{N}_m.\ x \neq y\ \&\ f(x) = f(y))$

1.5.3 The Formal Proof

$\vdash (\exists f{:}(\mathbb{N}_m \rightarrow \mathbb{N}_k).\ \text{Inj}(\mathbb{N}_m;\ \mathbb{N}_k;\ f)) \Rightarrow m{\le}k$

by Induction on m, with trivial base case $0{\le}k$

1. $m : \mathbb{Z}$
2. $0{<}m$
3. $\forall k'{:}\mathbb{N}.\ (\exists f'{:}(\mathbb{N}_{m-1} \rightarrow \mathbb{N}_{k'}).\ \text{Inj}(\mathbb{N}_{m-1};\ \mathbb{N}_{k'};\ f')) \Rightarrow m{-}1{\le}k'$
4. $k : \mathbb{N}$
5. $\exists f{:}(\mathbb{N}_m \rightarrow \mathbb{N}_k).\ \text{Inj}(\mathbb{N}_m;\ \mathbb{N}_k;\ f)$

$\vdash m{\le}k$ by Analyze'5

5. $f : \mathbb{N}_m \rightarrow \mathbb{N}_k$
6. Inj$(\mathbb{N}_m;\ \mathbb{N}_k;\ f)$

$\vdash m{\le}k$ by $m{-}1{\le}k{-}1$ Asserted .... THEN $k{-}1 \in \mathbb{N}$ Asserted

\
.....assertion .....

$\vdash k{-}1 \in \mathbb{N}$ by $f(0) \in \mathbb{N}_k$ Asserted . . . '
—
7. $k{-}1 \in \mathbb{N}$
$\vdash m{-}1{\le}k{-}1$ by BackThru: Hyp:3 . . . .

$\vdash \exists f'{:}(\mathbb{N}_{m-1} \rightarrow \mathbb{N}_{k-1})$. Inj($\mathbb{N}_{m-1}$; $\mathbb{N}_{k-1}$; f')

by Witness: Replace $k{-}1$ by $f(m{-}1)$ in f . . . w, etc.

$\vdash$Inj($\mathbb{N}_{m-1}$; $\mathbb{N}_{k-1}$; Replace $k{-}1$ by $f(m{-}1)$ in f)

by BackThru:
Thm* Inj($\mathbb{N}_{m+1}$; $\mathbb{N}_{k+1}$; f)
$\Rightarrow$
Inj($\mathbb{N}_m$; $\mathbb{N}_k$; Replace k by $f(m)$ in f) . . . ., etc.

1.5.4 Formal Proof of the Pigeonhole Principle

$\vdash \forall m,k{:}\mathbb{N}, f{:}(\mathbb{N}_m \rightarrow \mathbb{N}_k).\ k{<}m \Rightarrow (\exists x,y{:}\mathbb{N}_m.\ x \neq y\ \&\ f(x) = f(y))$ by Auto

1. $m : \mathbb{N}$
2. $k : \mathbb{N}$
3. $f : \mathbb{N}_m \rightarrow \mathbb{N}_k$
4. $k{<}m$

$\vdash \exists x,y{:}\mathbb{N}_m.\ x \neq y\ \&\ f(x) = f(y)$

by Inst:
Thm* $\forall m{:}\mathbb{N}, f{:}(\mathbb{N}_m \rightarrow \mathbb{Z})$. ¬Inj($\mathbb{N}_m$; $\mathbb{Z}$; f) $\Rightarrow (\exists x{:}\mathbb{N}_m, y{:}\mathbb{N}_x.\ f(x) = f(y))$
on
Tms:[m | f] . . . .

\
. . . . .antecedent . . . . .
$\vdash$ ¬Inj($\mathbb{N}_m$; $\mathbb{Z}$; f) by $\neg m{\le}k$ Asserted . . . w

5. $\neg m{\le}k$
$\vdash$ ¬Inj($\mathbb{N}_m$; $\mathbb{Z}$; f) by SimilarTo -1 . . . .

5. Inj($\mathbb{N}_m$; $\mathbb{Z}$; f)
$\vdash m{\le}k$ by BackThru: Thm* ($\exists f{:}(\mathbb{N}_m \rightarrow \mathbb{N}_k)$. Inj($\mathbb{N}_m$; $\mathbb{N}_k$; f)) $\Rightarrow m{\le}k$ . . . .

$\vdash \exists f{:}(\mathbb{N}_m \rightarrow \mathbb{N}_k)$. Inj($\mathbb{N}_m$; $\mathbb{N}_k$; f) by Witness: f . . . .

$\vdash$ Inj($\mathbb{N}_m$; $\mathbb{N}_k$; f) by SimilarTo: Hyp:5 . . . .
—
5. $\exists x{:}\mathbb{N}_m, y{:}\mathbb{N}_x.\ f(x) = f(y)$
$\vdash \exists x,y{:}\mathbb{N}_m.\ x \neq y\ \&\ f(x) = f(y)$ by SimilarTo: -1 . . . .

1.5.5 Formal Proof of the Counter-example Lemma

$\vdash \forall m{:}\mathbb{N},\ f{:}(\mathbb{N}_m \rightarrow \mathbb{Z}).\ \neg$Inj($\mathbb{N}_m$; $\mathbb{Z}$; f) $\Rightarrow$ ($\exists x{:}\mathbb{N}_m,\ y{:}\mathbb{N}_x.\ f(x) = f(y)$) by Auto

1. $m : \mathbb{N}$
2. $f : \mathbb{N}_m \rightarrow \mathbb{Z}$
3. $\neg$Inj($\mathbb{N}_m$; $\mathbb{Z}$; f)

$\vdash \exists x{:}\mathbb{N}_m,\ y{:}\mathbb{N}_x.\ f(x) = f(y)$

by Decide: $\exists x{:}\mathbb{N}_m,\ y{:}\mathbb{N}_x.\ f(x) = f(y) \in \mathbb{Z}$ .... THEN Analyze3

4. $\neg(\exists x{:}\mathbb{N}_m,\ y{:}\mathbb{N}_x.\ f(x) = f(y))$

$\vdash$ Inj($\mathbb{N}_m$; $\mathbb{Z}$; f) by Analyze ....

5. $a_1 : \mathbb{N}_m$
6. $a_2 : \mathbb{N}_m$
7. $f(a_1) = f(a_2)$

$\vdash a_1 = a_2$ by $\neg a_1 < a_2$ & $\neg a_2 < a_1$ Asserted .... THEN Analyze .... THEN Analyze4

\

8. $a_1 < a_2$

$\vdash \exists x{:}\mathbb{N}_m,\ y{:}\mathbb{N}_x.\ f(x) = f(y)$ by Witness: a_2 .... THEN Witness: a_1 ....

—

8. $a_2 < a_1$

$\vdash \exists x{:}\mathbb{N}_m,\ y{:}\mathbb{N}_x.\ f(x) = f(y)$ by Witness: a_1 .... THEN Witness: a_2 ....

1.6 Natural Language Proof Texts

Here are examples of automatic generation of natural language text from Nuprl proofs [65].

Thm 1.3 *For natural number a and positive natural n, $Div(a; n; a/n)$.*

Consider that a is a natural number and n is a positive natural. Using the rem_bounds_1 lemma, the goal becomes $n * (a/n) \leq a < n * (a/n + 1)$.

Equivalently, the goal can be rewritten as $n*(a/n) \leq a < n*(a/n+1)$. By the $add\_mono\_wrt\_le\_rw$ lemma, the goal becomes $n * (a/n) \leq a < n * (a/n + 1)$.

Applying the $add\_mono\_wrt\_lt\_rw$ lemma, we conclude $n * (a/n) \leq a < n * (a/n + 1)$. By the add_com lemma, the goal becomes $n * (a/n) \leq a < n * (a/n + 1)$. Applying the add_com lemma, the goal becomes $n * (a/n) \leq a < n * (a/n + 1)$.

Equivalently, the goal can be transformed to $n * (a/n) \leq a < n * (a/n + 1)$.

Qed.

Thm 1.4 *For natural number a and positive natural n, if $a \geq n$ then a rem $b = (a - n)$ rem n.*

Consider that a is a natural number, n is a positive natural and $a \geq n$. Using the rem_to_div lemma, we know a rem $n = (a - n)$ rem n. From the div_rec_case lemma, the goal becomes $a - (a/n) * n = a - n - ((a - n)/n) * n$.

Equivalently, the goal can be transformed to $a - ((a_n)/n + 1) * n = a - n - ((a - n)/n) * n$. The result follows trivially.

Qed.

1.7 Deduction Systems

The systems we are discussing are not only formal specification languages, they are logics; thus they include a formal notion of *proof*. More than that, they are *implemented logics,* in that there are computer programs that help generate these proofs. We call the collection of programs and data structures that implement these logics their *deduction systems.*

The deduction systems usually include parsers, type checkers, proof checkers, theorem provers, and a theory management database or *library*. Because the data structures and programs are implemented on computers, these elements of the deduction system are formal, and they constitute part of a *formal metalanguage.* To the extent that the programming language is supported by a verification system, there are elements of a formal *metalogic* present as well. This is prominent in the case of ACL2, which is a logic for Common Lisp, its implementation language. It is also prominent in MetaPRL, whose implementation language is O'Caml, and whose compiler is actually described in MetaPRL.

1.7.1 Common Elements of the Proof Systems

The proof systems are based on Gentzen's *sequents,* mentioned earlier. As we said, in HOL and Nuprl, sequents have similar structure:

$$H_1, \ldots, H_n \vdash G$$

This means roughly that hypotheses $H_1, \ldots, H_n$ imply the conclusion G. In PVS, there can be multiple conclusions,

$$H_1, \ldots, H_n \vdash G_1, \ldots, G_m.$$

Together the H_i must imply one of the G_j. (If $m = 0$ then the H_i are inconsistent.)

Inference rules have these forms, where S_i are sequents:

Bottom Up	Top Down
$\dfrac{S_1, \ldots, S_k}{S}$	$\dfrac{S \text{ by}}{S_1}$ $\vdots$ S_k

For example,

$\dfrac{H, B \vdash G \quad H \vdash A}{H, A \Rightarrow B \vdash G}$	$H, A \Rightarrow B \vdash G$ by 1. $H, B \vdash G$ 2. $H \vdash A$

Recall that in Nuprl a sequent written as above, with H_i and G as types, means roughly that given $a_i \epsilon\ H_i$ we have constructed a computable function g such that $g(a_1, \ldots, a_n) \epsilon\ G$ and moreover, if $a_i = a'_i$ in H_i, then $g(a'_1, \ldots, a'_n) = g(a_1, \ldots, a_n)$ in G.

1.7.2 Status of Proof

The proof engines of the deduction system are designed to help users show the truth of propositions. This can only be done by building sequent proofs or declaring a proposition true by reference to an external source, or by some oracle or by fiat. We prefer proof construction.

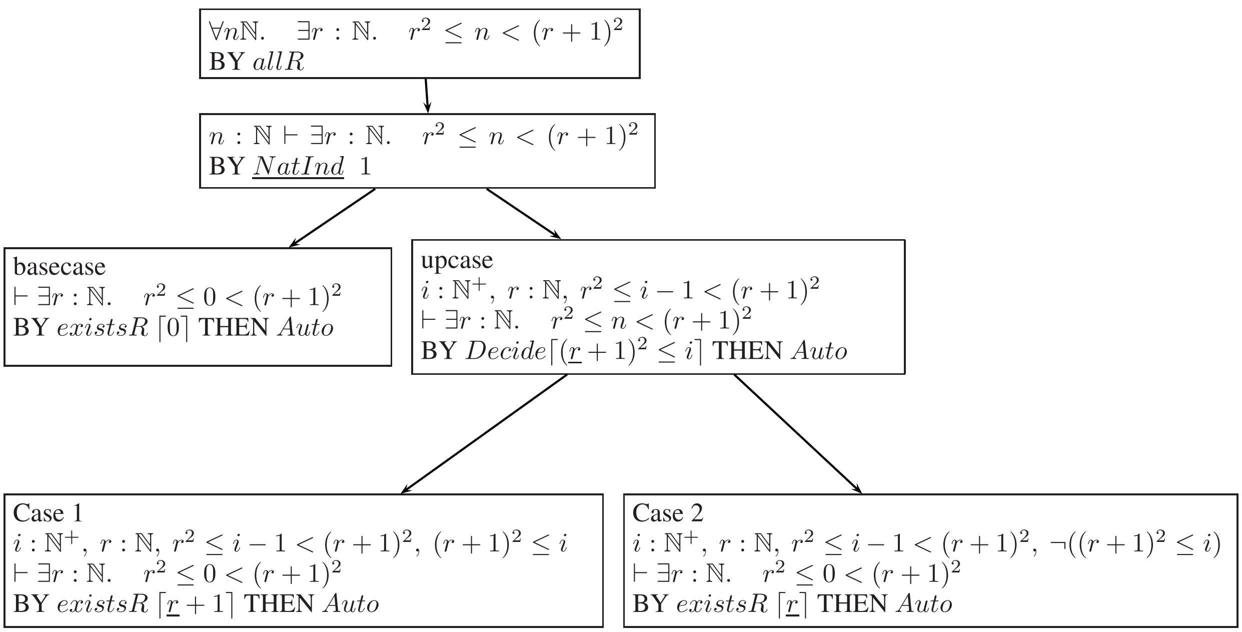
Proofs are part of the deductive system, but in these logics they are not terms of the logical language. However, Nuprl's *deductive system* includes a mapping from proofs into terms of the logic; the mapping is an *extractor*. There are two kinds of proof in the Nuprl deductive system:

1. Primitive proofs
2. Tactic tree proofs

In HOL, proofs are abstract. In PVS, there is a data structure in the Deduction System called a *proof script*, which is a linearized tree of tactics and rules.

Let us examine proofs of Thm 1.1. Figure 1.1 shows the Nuprl proof, a tactic tree. See also Figure A.1.

Figure 1.1: Tree-Form Proof of the Specification Theorem using Standard Induction.



In PVS the proof could be done in essentially the same way, but it is not displayed as a tree.

In HOL the proof is a bit longer, because there is no decision procedure for basic arithmetic; instead, various lemmas would be involved to explicitly prove the required assertions.

Chapter 2

Relating Logics

2.1 Comparing Logics

How might HOL, Nuprl and PVS share definitions and theorems — that is, could each theory consistently use a result of the other as an external source in proving one of its theorems?

Consider this example: In HOL and PVS there is a theorem that says that if any relation P is true of some number n, then there is a least n such that $P(n)$ is true. This is the least number principle (LNP).

LNP $\forall P : N \rightarrow \mathbb{B}.\ (\exists y : \mathbb{N}.\ P(y) \Rightarrow \exists x : \mathbb{N}.\ (P(x) \quad \& \quad \forall z : \mathbb{N}.\ z < x \Rightarrow \neg P(z)))$

If we simply translated this theorem into Nuprl, it would be provable, but it would mean something very different. The predicate P would be *decidable,* since it is a computable mapping from $\mathbb{N}$ into $\mathbb{B}$. As a result, we can effectively find the least number by a simple program:

```
least (P, y) =
  for i = 0 to y do
     if P(i) then return(i)
     else i := i + 1
  end; return(y).
```

But in HOL and PVS, P is any unary predicate, such as $P(x, m)$ iff a Turing machine with x states can print the number m and halt, starting with blank tape. It is not decidable what the least x is for every m. So Nuprl can't prove LNP for all of these m, but HOL and PVS can. Thus HOL and PVS claim that there is such a least number, regardless of whether or not we can find it.

In Nuprl, the full LNP is stated this way:

LNP $\forall P : \mathbb{N} \rightarrow Prop_i.\ (\exists y : \mathbb{N}.\ P(y) \Rightarrow \exists x : \mathbb{N}.\ (P(x) \quad \& \quad \forall z : \mathbb{N}.\ z < x \Rightarrow \neg P(x)))$

This theorem is not provable in Nuprl. However, if we added the Law of Excluded Middle to Nuprl, then we could prove the LNP.

Law of Excluded Middle (LEM) $\forall P : Prop_i.\ (P \vee \neg P)$

This has the effect of asserting that *all propositions are decidable,* and renders the computation system Omniscient. Indeed, to accept this axiom is to postulate some member of this type — we could call it *Omniscience* or *Magic.*

Adding this law would create a theory that we call *Classical Nuprl*. However, it might make Nuprl inconsistent. Indeed, Nuprl with domains is inconsistent with LEM [42]. But Doug Howe [67] showed that core Nuprl is consistent with LEM, and we will examine this result.

So how can we reconcile such divergent logics? In the early days of logic, the proponents of constructive principles were highly critical of classical mathematics and wanted to bring it crashing down. Typical of this period were the so-called "frog and mouse" wars, between the classical mathematician David Hilbert and the Intuitionist, L. E. J. Brouwer [113].

There were also attempts at reconciliation and mutual understanding. For example, Gödel showed how to interpret classical number theory inside constructive number theory. He showed how to define the classical logical connectives and quantifiers in terms of the constructive ones. For example,

$$A \circledvee B == \neg(\neg A \,\&\, \neg B)$$

$$Ey : A.\ P == \neg\forall y : A.\ (\neg P)$$

He translated atomic predicates P into $\neg\neg P$.

These ideas were extended by Harvey Friedman [50] to type theories such as the Simple Theory of Types. Thus, there is a sense in which much of HOL and PVS could be translated into Nuprl. Indeed, they can be translated fully into Coq.

Taking this approach essentially means adding more logical operators and thus complicating matters because of all the intermixing of operators made possible.

Bishop [30] suggested a different and more pragmatic approach. He proposed doing constructive mathematics in the style that all constructive theorems could also be read classically. He would cite core Nuprl as a good example. The classical mathematician or computer scientist can read Nuprl as it stands, thinking of $Prop$ as a synonym for *bool*, taking $A \rightarrow B$ as *all* functions from A to B, and reading the logical operators classically. All that would seem strange in this world is that users of Nuprl would develop more involved proofs, and would not prove certain standard theorems. If we allow the option of LEM — Classical Nuprl — then both sides can work in the same space. We have tried to work in the manner and style of Bishop.

In the case of HOL as defined by Gordon and Melham, it is not possible to follow Bishop because the logic is inherently classical and incompatible with constructive interpretation. This is because $\exists x : A.\ P$ is defined using the choice operator and $\forall x : A.\ P(x)$ is defined from the existential, as we have seen. However, as Pavel Naumov discovered to our immense delight, the actual HOL system takes *all* the logical operators as primitive. Pavel shows how to find constructive content in HOL proofs.

PVS also defines its logical operators rather than making them primitive. We have seen that $\forall x : A.\ P(x)$ means $\lambda x : A.\ P(x) = \lambda x : A.\ T$. The Boolean operators are defined normally, and as with HOL, there is no type $Prop$ that can be used instead of $\mathbb{B}$.

Our task of relating theories would be simpler if HOL and PVS had accommodated a possible constructive interpretation. This could be done by adding an additional layer of definition that introduced the type $Prop$ and the full set of logical operators: $\&, \vee, \Rightarrow, \neg, \forall, \exists$. It would then be possible to state theorems or axioms that related these to the Boolean operations. When these theorems are not used, then a constructive interpretation is possible.

In the Bishop approach, we see classical mathematics arising from the introduction of an oracle (or several oracles) into the computational interpretation. To make this rigorous, we need a semantics of such computations. Howe's work and related semantics by Dybjer offer one such semantics, but it is fully classical, as we'll see.

There is a direct complement to Bishop's approach. The idea is to interpret Nuprl fully classically and tolerate the additional syntax of $Prop$ as a recognition that logic is more fundamental than Boolean algebra. Logic in fact deals with propositions and is more faithfully captured by first-order logic than by Boolean functions. The constructive aspect of Nuprl is then seen as an "overlay;" a way to treat certain subsets of the logic in a computationally meaningful way.

For this approach to make sense, there must be a coherent classical semantics for Nuprl. This is precisely what we turn to it next.

2.2 Set-Theoretic Interpretations of Type Theory

The standard interpretation of HOL is given in terms of sets up to level $\omega + \omega$ in the cumulative hierarchy — starting with the empty set, $\emptyset$, and forming power sets, $P(\emptyset), P(P(\emptyset), \ldots$ and taking unions at the limit ordinals. At stage ω we take the union of all the finite power sets. As mentioned earlier (Section 1.4), Andy Pitts provides a very clear and detailed account in the book *Introduction to HOL,* by Gordon and Melham [51].

Can we find a similar set theoretic interpretation for Martin-Löf type theories, including Nuprl? Can the methods used for HOL be applied to PVS? That topic is taken up by Owre and Shankar [98], who give a qualified positive reply. The ideas used for Martin-Löf type thoeries play an important role in clarifying the situation for PVS, so we will focus on them; for instance, those ideas completely settle the case of dependent types of monomorphic objects (those with unique types), as is the case for PVS.

Already in 1987, Troelstra [110] pointed out that Martin-Löf type theory without universes, ML_0, could be interpreted in classical set theory in which the dependent function space was simply the set theoretic cartesian product $\Pi x \in A.B = \{f : A \to \sigma x \in A.B \mid \forall x : A.f(x) \in B(x)\}$ with extensional equality.

In 1990, Peter Dybjer [45] showed how to extend Troelstra's observation to universes and a schema for inductive sets and give a classical semantics to the intensional, monomorphic type theory of Martin-Löf 1986 as presented in the monograph of Nordström, Peterson and Smith [95].

> The basic idea is to interpret a type-theoretic concept as the corresponding set-theoretic concept, which usually has the same name. So a (type-theoretic) set is interpreted as a (set-theoretic) set, an element of a set as an element of a set, (definitional) equality as (extensional) equality, (type-theoretic) cartesian product as (set-theoretic) cartesian product, function as function graph, etc. A context is interpreted as a set of assignments.
>
> – *Peter Dybjer* [6, 7, 16, 44, 95, 101, 110]

2.2.1 Interpreting Expressions

Dybjer uses $[\![a]\!]\rho$ as the denotation of the expression a under the assignment ρ. He assigns a set to each variable in a finite list of variables which includes all variables which are free in a. Let $\emptyset$ be the empty assignment and let ρ_x^u abbreviate $\rho \cup \{\langle x, u\rangle\}$. Let also $[\![a]\!]$abbreviate $[\![a]\!]\emptyset$.

The interpretation function is partial. Partiality is introduced to treat application; however, the interpretation of a derivable judgment will always be defined and true. (The method with a partial interpretation function has also been used by Streicher for a categorical interpretation of the calculus of constructions [107].)

Dybjer defines the function space as follows:

$$[\![\Pi x : A_0.A_1[x]]\!]\rho = \prod_{u \in [\![A_0]\!]\rho} [\![A_1[x]]\!]\rho_x^u.$$

This is defined iff $[\![A_0]\!]\rho$ is defined and $[\![A_1[x]]\!]\rho_x^u$ is defined whenever $u \in [\![A_0]\!]\rho$.

Variables are defined as:

$$[\![x]\!]\rho = \rho(x),$$

and functions as:

$$[\![\lambda x : A.a[x]]\!]\rho = \{\langle u, [\![a[x]]\!]\rho_x^u\rangle \mid u \in [\![A]\!]\rho\}.$$

This is defined iff $[\![A]\!]\rho$ is defined and $[\![a[x]]\!]\rho_x^u$ is defined whenever $u \in [\![A]\!]\rho$.

Application is defined as:

$$[\![a_1(a_0)]\!]\rho = ([\![a_1]\!]\rho)([\![a_0]\!]\rho).$$

Application is defined iff $[\![a_1]\!]\rho$ and $[\![a_0]\!]\rho$ are defined, and $[\![a_1]\!]\rho$ is a function the domain of which *contains* $[\![a_0]\!]\rho$. (Notice that it is possible to interpret polymorphic application in set theory using this liberal notion of domain. This is not the case for all interpretations of type theory; compare Streicher [107].)

Interpretation of context expressions is defined as:

$$[\![\epsilon]\!] = \{\emptyset\}.$$

This is always defined.

$$[\![\Gamma, x : A]\!] = \{\rho_x^u \mid \rho \in [\![\Gamma]\!] \wedge u \in [\![A]\!]\rho\}.$$

This is defined iff $[\![\Gamma]\!]$ is defined and $[\![A]\!]\rho$ is defined whenever $\rho \in [\![\Gamma]\!]$.

Judgments are defined as:

$$[\![\Gamma context]\!] \quad \text{iff} \quad [\![\Gamma]\!] \text{ is a set of assignments.}$$

This is defined iff $[\![\Gamma]\!]$ is defined.

$$[\![\Gamma \vdash A\ set]\!] \quad \text{iff} \quad [\![A]\!]\rho \text{ is a set whenever } \rho \in [\![\Gamma]\!].$$

This is defined iff $[\![\Gamma]\!]$ is defined and if $[\![A]\!]\rho$ is defined whenever $\rho \in [\![\Gamma]\!]$.

$$[\![\Gamma \vdash a : A]\!] \quad \text{iff} \quad [\![a]\!]\rho \in [\![A]\!]\rho \text{ whenever } \rho \in [\![\Gamma]\!].$$

This is defined iff $[\![\Gamma]\!]$ is defined and if $[\![a]\!]\rho$ and $[\![A]\!]\rho$ are defined whenever $\rho \in [\![\Gamma]\!]$.

$$[\![\Gamma \vdash A = A']\!] \quad \text{iff} \quad [\![A]\!]\rho = [\![A']\!]\rho \text{ whenever } \rho \in [\![\Gamma]\!].$$

This is defined iff $[\![\Gamma]\!]$ is defined and if $[\![A]\!]\rho$ and $[\![A']\!]\rho$ are defined whenever $\rho \in [\![\Gamma]\!]$.

$$[\![\Gamma \vdash a = a' : A]\!] \quad \text{iff} \quad [\![a]\!]\rho = [\![a']\!]\rho \wedge [\![a]\!]\rho \in [\![A]\!]\rho \text{ whenever } \rho \in [\![\Gamma]\!].$$

This is defined iff $[\![\Gamma]\!]$ is defined and if $[\![a]\!]\rho,\ [\![a']\!]\rho$, and $[\![A]\!]\rho$ are defined whenever $\rho \in [\![\Gamma]\!]$.

2.2.2 Soundness of the Inference Rules

An inference rule is sound if the interpretation of the conclusion of a rule is defined and true whenever the interpretation of the premises are defined and true. It is routine to check the soundness of all the inference rules. As an illustration Dybjer shows the soundness of the rule of application. The premises are interpreted as

$$[\![a_1]\!]\rho \in \prod_{u \in [\![A_0]\!]\rho} [\![A_1[x]]\!]\rho_x^u \text{ whenever } \rho \in [\![\Gamma]\!]$$

and

$$[\![a_0]\!]\rho \in [\![A_0]\!]\rho \text{ whenever } \rho \in [\![\Gamma]\!].$$

From this he concludes that

$$([\![a_1]\!]\rho)([\![a_0]\!]\rho) \in [\![A_1[x]]\!]\rho_x^{[\![a_0]\!]\rho} \text{ whenever } \rho \in [\![\Gamma]\!],$$

and hence the conclusion of the rule follows, since

$$[\![A_1[x]]\!]\rho_x^{[\![a_0]\!]\rho} = [\![A_1[a_0]]\!]\rho$$

follows from a substitution lemma which holds for the interpretation.

2.3 Howe's Semantics

2.3.1 Overview

Howe's idea is to map types and terms of Nuprl into a standard model of sets — the cumulative hierarchy. To deal with Nuprl's universes, the hierarchy of sets has to "go very high" into the inaccessible cardinals (Dybjer suggested this as well).

The basic strategy is to create a term model of set theory of very high cardinality and add the terms of type theory to it, those for types and their elements. This model is called T_0. Howe then adds an evaluation relation on T_0, $a \Downarrow a'$; this is a generalization of the Nuprl evaluation relation to T_0. Of course, on set terms it is no longer an effective operation; yet on type terms it is. This is a significant departure from Dybjer and others.

Howe also adds an approximation relation to T_0 which relates sets and terms; he says that a set α *approximates* a type term a (or a *subsumes* α or α *covers* a), written $\alpha \lhd a$. The idea is that a graph of a function ϕ approximates a polymorphic function term, $\lambda x.\ b$ precisely when for all pairs $< \alpha, \beta >$ in ϕ, β approximates $b[\hat{\alpha}/x]$, where $\hat{\alpha}$ is the term in T_0 corresponding to the set α.

There are two subtle parts of Howe's model, but neither of them is needed to understand the connections to HOL and PVS, so we will not pursue them here. One issue is justifying Nuprl's direct computation rules in the new semantics. This topic appeals to Doug because his brilliant insights about computational pre-orders made these rules possible. However, neither HOL nor PVS has these rules, and they are in principle dispensable in Nuprl. For example, they are not in Martin-Löf type theory nor in Alf, nor in Coq.

The other subtle point in Howe's model is his elegant treatment of quotient types, $A//E$. Neither HOL, PVS, Coq, nor Martin-Löf type theory uses quotients at present, although they are exceedingly useful and mathematically elegant.

So we are left with the treatment of *polymorphic functions* and *ordered pairs* as the main elements of Howe's model, and these are the easiest parts to master. The model supports a simple semantics for sequents, and it allows us to easily show that the Nuprl rules are sound. His model also shows the soundness of HOL rules, and this allows us to directly relate HOL and core classical Nuprl, showing that they are relatively consistent. A similar argument would apply to a fragment of PVS, but more work is required to model the PVS theory, and we are not the best people to do this while the system is still evolving.

2.3.2 Cumulative Hierarchy of Sets and Tagged Sets

Let $Z_0 = \emptyset$, the empty set, and $Z_{\sigma+1} = Pow(Z_\sigma)$, the power set of Z_σ for σ an ordinal. Let $Z_\tau = U_{\sigma<\tau} Z_\sigma$ for τ a limit ordinal. Let σ_0 be the limit of a countable sequence of inaccessible cardinals, $\tau_1 < \tau_2 < \tau_3 < \cdots$. Take $Z = U_{\tau<\sigma_0} Z_\tau$ as the segment of the cumulative hierarchy we need to model Nuprl. For HOL we only need $w + w$.

For any set α in Z, its rank, denoted $rank(\alpha)$, is the least ordinal $\tau < \sigma_0$ such that $\alpha \epsilon Z_\tau$.

We distinguish several kinds of sets based on their structure, and we tag them. For instance, the Nuprl natural numbers are $0, 1, 2, \ldots$. They can be mapped into Z sets in the standard way, say $num(0) = \emptyset$, $num(1) = \{\emptyset\}$, $num(2) = \{\emptyset, \{\emptyset\}\}$, $num(i+1) = num(i) \cup \{num(i)\}$. We will tag these sets with num, forming $< num, \emptyset >$, $num, \{\emptyset\} >$, etc. Ordered pairs, $< a, b >$, are represented by $\{a, \{a, b\}\}$. We take these as $< pair, \{a, \{a, b\}\} >$. Functions are single valued sets of ordered pairs, i.e. $< x, y >, <$

$x', y' > \epsilon\phi$ and $x = x'$ implies that $y = y'$. We tag these as $< fun, \phi >$. A subset of tagged elements is a possible type, and we tag them ty, γ. Summarizing, the tagged elements are $W \subset Z$ such that:

1. $< ty, \gamma > \epsilon W$ if $\gamma \subset W$
2. $< fn, \phi > \epsilon W$ if $\phi \subset W \times W$ and is single-valued
3. $< c_i, < x_1, \ldots, x_n >> \epsilon W$ if $x_i \epsilon W$ and c_i is a constructor such as num, $pair$, etc.

2.3.3 Encoding Types and Terms

Howe establishes by definition a *unique meaning property* of the set model. He wants there to be a unique set γ_A that encodes or approximates a type A, and given γ_A he wants that for each $a \epsilon A$, there is a unique set α of γ_A that approximates a. However, this is not a property of W as it stands. For example, the functions $\phi_1 = \{< 0, 0 >\}$ and $\phi_2 = \{< 1, 1 >\}$ are both approximations to the identity function, $\lambda x.x$, i.e. $\phi_1 \lhd \lambda x.x$ and $\phi_2 \lhd \lambda x.x$. When we know the type of $\lambda x.x$, say $\{x : \mathbb{N} | x = 0\} \to \{x : \mathbb{N} | x = 0\}$, then we know which function belongs to this type. We do not want to allow a set like $\{\phi_1, \phi_2\}$ to be in the model to represent a type. Doug calls such elements *consistent.*

Definition Consistency between two elements $x, y \epsilon W, con(x, y)$, is defined by induction on rank as follows:

- $con(\gamma, \gamma)$.
- $con(\phi_1, \phi_2)$ if for all $< \alpha_1, \beta_1 > \epsilon\phi_1$ and $< \alpha_2, \beta_2 > \epsilon\phi_2$, if $con(\alpha_1, \alpha_2)$ then $con(\beta_1, \beta_2)$.
- $con(c_i(x_1, \ldots, x_n), c_i(x'_i, \ldots, x'_n))$ if for all pertinent j, $con(x_j, x'_j)$.

Definition Define $V \subset W$ by rank induction:

- $\gamma \epsilon V$ if $\gamma \subset W$ and for all $\alpha, \alpha' \epsilon \gamma$, $con(\alpha, \alpha')$ implies $\alpha = \alpha'$.
- $\phi \epsilon V$ if $\phi \subset V \times V$ and $con(\phi, \phi)$.
- $c_i(x_1, \ldots, x_n) \epsilon V$ if $x_j \epsilon V$ for $1 \leq j \leq n$.

We can now build a term model using V.

2.4 A Term Language, T_0

We will now build a term language that includes constants for the sets in V of kind γ and ϕ; in addition we add notations for types that require binding, $x : A \to B$, $x : A \times B$, $\{x : A | B\}$ as well as lambda terms $\lambda x.b$, applications $f(a)$ where f and a are terms, and the operators $c_i(a_1, \ldots, a_n)$ for a_i terms and c_i the Nuprl constructors such as $pair$, $number$, $void$, inr, inl, and so forth.

For every set $\alpha \epsilon V$ there is a unique term $\hat{\alpha}$ in T_0. The sets of the form $< c_i, < \alpha_1, \ldots, \alpha_n >>$ are represented by the term $c_i(\hat{\alpha}_1, \ldots, \hat{\alpha}_n)$. The other sets of V are represented by the corresponding constant.

We could use a countable term language if we started with a countable model of ZFC plus inaccessible cardinals, but for Howe's results this is not necessary.

Next, we introduce an "operational" semantics on terms by giving rules for *evaluation,* $a \Downarrow a'$, and rules for *approximation* of a term of a of T_0 by a set α of V, $\alpha \lhd a$.

Howe shows next a uniqueness result that he calls *coherence.*

Figure 2.1: Evaluation Rules

$$\frac{f \Downarrow \hat{\phi} \quad (\alpha, \beta) \in \phi \quad \alpha \lhd a}{f(a) \Downarrow \hat{\beta}} \ (ap_\phi) \qquad \frac{f \Downarrow \lambda x.b \quad b[a/v] \Downarrow v}{f(a) \Downarrow v} \ (ap_\lambda)$$

$$\frac{}{\hat{\alpha} \Downarrow \hat{\alpha}} \qquad \frac{}{\lambda x.b \Downarrow \lambda x.b} \qquad \frac{}{c_i(\overline{a}) \Downarrow c_i(\overline{a})}$$

Figure 2.2: Approximation Rules

$$\frac{e \Downarrow v \quad \alpha \lhd v}{\alpha \lhd e} \qquad \frac{\forall j \quad \alpha_j \lhd a_j}{c_i(\alpha_1, \ldots, \alpha_n) \lhd c_i(\hat{\alpha}_1, \ldots, \hat{\alpha}_n)}$$

$$\frac{\forall (\alpha, \beta) \in \phi \quad \beta \lhd b[\hat{\alpha}/x]}{\phi \lhd \lambda x.b}$$

Thm 2.1 *For any term t in T_0,*

1. *If $\gamma_1 \lhd t$ and $\gamma_2 \lhd t$ then $\gamma_1 = \gamma_2$*
2. *For all $\gamma \in V$ and $\alpha_1, \alpha_2 \in \gamma$,*
   if $\alpha_1 \lhd t$ and $\alpha_2 \lhd t$ then $\alpha_1 = \alpha_2$

The proof is by induction on the definitions of $\Downarrow$ and $\lhd$.

This theorem will allow us to assign a unique set theoretic term to each term t in T_0 that is a type, and given a type A and a term of that type, to assign a unique set that approximates it (or encodes it).

2.5 Semantics of Core Types

A notion fundamental to this semantics is that the meaning of a term is given by finding its value. If t evaluates to a set, then t is a type. If a evaluates to a term that belongs to a type A, then $a \in A$. The types will be closed terms A that evaluate to set constants $\hat{\gamma}$. If a is a type, then let $[\![A]\!]$ denote γ. A closed term a belongs to a type A exactly when it evaluates to a term a' that is approximated by a set α in A, let $[\![a]\!]_A = \alpha$.

Here are examples of how we assign meaning to Nuprl terms.

1. The empty type.

   The empty type, *void,* one of the c_i, is encoded as the empty set $\emptyset$, i.e. $[\![void]\!] = \emptyset$.

2. The natural numbers

   The term $\mathbb{N}$ is one of the constructors, c_i. We say that $[\![\mathbb{N}]\!] = \omega$. The meaning of the elements is given inductively, $[\![0]\!]_{\mathbb{N}} = \emptyset$, $[\![n+1]\!]_{\mathbb{N}} = [\![n]\!]_{\mathbb{N}} \cup \{[\![n]\!]_{\mathbb{N}}\}$. We also say that $n \Downarrow [\![n]\!]_{\mathbb{N}}$.

3. Dependent function spaces

   Suppose that term A evaluates to a set constant $\hat{\gamma}$, $A \Downarrow \hat{\gamma}$, and for each $\alpha \in \gamma$, the term $B[\hat{\alpha}/x]$ evaluates to the set constant $\hat{\gamma}_\alpha$. Then $x : A \to B$ evaluates to the set $\{\phi \in V|$ the domain of ϕ is γ and for each $< \alpha, \beta > \in \phi, \beta \in \gamma_\alpha\}$, which we denote as $[\![x : A \to B]\!]$.

4. Elements of function spaces

   A term f belongs to $x : A \to B$ if there is a function term ϕ in $[\![x : A \to B]\!]$ such that $\phi \lhd f$. Note that $\lambda x.b \in x : A \to B$ exactly when there is a $\phi \in [\![x : A \to B]\!]$ such that $\phi \lhd \lambda x.b$.

5. Dependent product

   Suppose that $A \Downarrow \hat{\gamma}$ and for each $\alpha \in \gamma$, $B[\hat{x}/x] \Downarrow \hat{\gamma}_\alpha$. Then $x : A \times B \Downarrow \{< pair, < \alpha, \beta >> | \alpha \in \gamma$ and $\beta \in \gamma_\alpha\}$. We write this as $[\![x : A \times B]\!]$.

6. Ordered pairs

   A term p belongs to $x : A \times B$ if there is a set term δ in $[\![x : A \times B]\!]$ such that $\delta \lhd p$. This δ will be tagged as a constructor $< pair, < \alpha, \beta >>$. We thus know that p must be the term $pair(\hat{\alpha}, \hat{\beta})$, and $[\![pair(\hat{\alpha}, \hat{\beta})]\!]_{x:A \times B} =< pair, < \alpha, \beta >>$.

2.6 Semantics of Core Nuprl

The set theoretic semantics defined so far will apply equally well to HOL and PVS. What distinguishes Nuprl from them in this setting is the polymorphic nature of its functions, the subtype relation, $A \sqsubseteq B$, that is derived from the polymorphism, and the existence of quotient types.

To accommodate these Nuprl specific ideas, Howe needs two other concepts. One is a more refined approximation notion for functions based on capturing part of the computational preorder on Nuprl terms. Howe denotes this order by $\leq$. We define it below after we discuss quotients because this order on elements of quotient types is critical.

2.6.1 Quotients

Nuprl directly captures Frege's fundamental means of abstraction, equivalences relations on type. Here are two examples — the rational numbers and the integers modulo n.

A standard way to define the rational numbers, $\mathbb{Q}$, is to first build the *fractions,* $< nat, d >$ where $n \in \mathbb{Z}$ and $d \in \{z : \mathbb{Z} | z \neq 0\}$. Two fractions, $< n_1, d_1 >, < n_2, d_2 >$ are equal just when their "cross products" are equal, i.e. $n_1 * d_2 = n_2 * d_1$. Thus, $< 2, 3 >=< 8, 12 >$ since $2 * 12 = 3 * 8$. The rationals are the fractions with this equivalence relation. The notation we use is:

$$(\mathbb{Z} \times \{z : \mathbb{Z} | z \neq 0\})//\lambda x, y.(\mathit{1of}(x) * \mathit{2of}(y) = \mathit{lof}(y) * \mathit{2of}(x)).$$

It is easy to build the algebraic operations on the fractions, e.g.

$$< n_1, d_1 > * < n_2, d_2 >=< n_1 * n_1, d_1 * d_2 >,$$

and show that they respect the equivalence relation. We see the equivalence relation as hiding detail and thus raising the level of abstraction.

In general, if A is a type and E is an equivalence relation on E, then we let $A//E$ denote the quotient type of A by E. In Nuprl the effect of the quotient is to simply impose a new equality on the type A. This is illustrated well by the type of integers modulo n, the congruence integers. Let $\mathbb{Z}_n$ be the integers mod n. For example, in $\mathbb{Z}_2$, $0 = 2 = 4 = \cdots$ and $1 = 3 = 5 = \cdots$. Set theoretically we think of the elements of $\mathbb{Z}_2$ as equivalence classes, say

$$[0] = \{z : \mathbb{Z} | z = 0 \text{ mod } 2, \ [1] = \{z : \mathbb{Z} | z = 1 \text{ mod } 2.$$

Thus, $\mathbb{Z}_2$ is $\{[0], [1]\}$.

Howe's account of $A//E$ requires that we introduce a tern $[a]$ to denote the elements, but these are not equivalence classes, just tagged elements of A. We have for $\mathbb{Z}_2$ that $[0] = [2] = [4] \cdots$. The set term for $A//E$ will use equivalence classes to denote these elements.

2.6.2 Subtyping in Nuprl

For types A and B, we say that $A \sqsubseteq B$ if and only if $a_1 = a_2$ in A implies that $a_1 = a_2$ in B. This means that a_1 and a_2 are in both A and B. For example, $\mathbb{Z}_6 \sqsubseteq \mathbb{Z}_2$ because $[a_1] = [a_2]$ in $\mathbb{Z}_6$, say $[0] = [6]$, implies that $[a_1] = [a_2]$ in $\mathbb{Z}_2$, i.e. $[0] = [6]$ mod 2.

In standard Nuprl, we have $A \sqsubseteq A//E$ since the notations $[a]$ are not used. Howe would write $\mathbb{Z}$ as $\mathbb{Z}_0$, then $\mathbb{Z}_0 \sqsubseteq \mathbb{Z}_k$, for $k > 0$. For any type A there is the default quotient, $A//\lambda x, y.\ (x = y \text{ in } A)$, which we could call A_0, then $A_0 \sqsubseteq A//E$.

The *extensional equality* relation on types is defined as $A \equiv B$ iff $A \sqsubseteq B$ and $B \sqsubseteq A$.

The polymorphic nature of functions means that if $A \sqsubseteq A'$ and $B \sqsubseteq B'$, then $A' \to B \sqsubseteq A \to B$. For example,

$$(\mathbb{Z}_2 \to \mathbb{B}) \sqsubseteq (\mathbb{Z}_6 \to \mathbb{B}).$$

This is because a function $f \in \mathbb{Z}_2 \to \mathbb{B}$ has the property that $x = y$ *mod* 2 implies $f(x) = f(y)$ in $\mathbb{B}$, and each element of $\mathbb{Z}_6$, say $[n]$, is also an element of $\mathbb{Z}_2$; thus $f([n])$ is defined, and $x = y$ *mod* 6 implies $x = y$ *mod* 2, thus $f(x) = f(y)$ for elements of $\mathbb{Z}_6$.

2.6.3 Semantics of Polymorphism

The ordinary set theory semantics given for HOL does not support Nuprl polymorphism. Given an HOL function from $\mathbb{Z}_2$ into $\mathbb{B}$, it is a specific set of ordered pairs, say $\phi_1 = \{< [0], t >, < [1], f >\}$. A function from $\mathbb{Z}_6$ into $\mathbb{B}$ looks like this:

$$\phi_2 = \{< [0], t >, < [1], f >, < [2], t >, < [3], f >, < [4], t >, < [5], f >\}.$$

The ϕ_1 graph is not an element of $\mathbb{Z}_6 \to \mathbb{B}$, it is far "too small."

To capture the full polymorphic meaning of $\lambda x.b$ in Nuprl, we need another notion, part of the computational preorder. We define it for a set representation of equivalences classes as well. These are sets whose elements are equivalences classes, and they are tagged by ξ.

Definition For α, β in V define the following preorder, $\alpha \leq \beta$, by induction on the rank of α:

1. $\gamma \leq \gamma$.
2. $c_i(\alpha_1, \ldots, \alpha_n) \leq c_i(\alpha'_1, \ldots, \alpha'_n)$ if $\alpha_j \leq \alpha'_j \quad 1 \leq j \leq n$.
3. $\phi \leq \phi'$ if for all $< \alpha, \beta > \in \phi$, there exist $< \alpha', \beta' > \in \phi'$ such that $\alpha' \leq \alpha$ and $\beta \leq \beta'$ (note the contravariance in the first argument).
4. $\xi \leq \xi'$ if for all $\alpha' \in \xi'$ there is $\alpha \in \xi$ with $\alpha \leq \alpha'$.

We now incorporate $\leq$ into the approximation relation, concluding $\alpha \lhd \beta$ when we know $\alpha \leq \beta$.

After this example we will prove a key lemma that asserts that if $\psi \leq \phi$ and $\phi \lhd \lambda x.b$, then $\psi \lhd \lambda x.b$. We illustrate the value of the result as follows.

Suppose $\lambda x.b \in \mathbb{Z}_2 \to \mathbb{B}$ because $[\![\lambda x.b]\!] = \phi \in [\![\mathbb{Z}_2 \to \mathbb{B}]\!]$. Can we show that $\lambda x.b \in \mathbb{Z}_6 \to \mathbb{B}$ by finding an element of $\psi \in [\![\mathbb{Z}_6 \to \mathbb{B}]\!]$ that $\psi \lhd \lambda x.b$?

We know essentially that if $\phi_1 \leq \phi_2 \lhd \lambda x.b$ then $\phi_1 \lhd \lambda x.b$. Can we find ψ so that $\psi \leq \phi$?

if $< \alpha, \beta > \in \psi$ then $\exists < \alpha', \beta > \in \psi$
such that $\alpha' \leq \alpha \quad (\beta = \beta)$
  note $< \alpha_1 \beta > \in \psi$ means $\alpha \in \mathbb{Z}_6$, can we
  find $\alpha' \in \mathbb{Z}_2$ such that $\alpha' \leq \alpha$ YES,

e.g. $z \in \mathbb{Z}_6 \quad z = \{0, 6, 12, \cdots\}$ then
$e = \{0, 2, 4, \cdots\}$ gives $e \leq z$
so for $\alpha = z$ take $\alpha' = e$

So since $\psi \leq \phi$ and $\phi \lhd \lambda x.b$, we have $\psi \lhd \lambda x.b$ thus $[\![\lambda x.b]\!]_{\mathbb{Z}_6 \to \mathbb{B}} = \psi$.
For example, take $\lambda x.even(x)$ as a concrete case.

Thm 2.2 *If $\alpha \leq \beta$ and $\beta \lhd e$ then $\alpha \lhd e$.*

To prove this theorem, we use a subrelation of approximation, $\alpha \sqsubset e$, which holds iff there is a term v such that $e \Downarrow v$ and $\alpha \sqsubset v$. We define $\sqsubset$ inductively on rank as follows.

Definition For $\alpha \in V$ and $e \in T_0$, we say:

1. $\alpha \sqsubset e$ if there exists β with $e \Downarrow \hat{\beta}$ and $\alpha \leq \beta$.
2. $\phi \sqsubset e$ if $e \Downarrow \lambda x.b$ and for all $< \alpha, \beta > \in \phi$ and all terms such that $\alpha \sqsubset a$, $\beta \sqsubset b[a/x]$.
3. $\xi \sqsubset e$ if $e \Downarrow [a]$ and there is $\alpha \in \xi$ such that $\alpha \sqsubset a$.
4. $c_i(\alpha_1, \ldots, \alpha_n) \sqsubset e$ if $e \Downarrow c_i(a_1, \ldots, a_n)$ and $\alpha_i \sqsubset a_i \quad 1 \leq i \leq n$.

It is easy to show.

Lemma 2.1 *If $\alpha \leq \alpha'$ and $\alpha' \sqsubset e$ then $\alpha \sqsubset e$.*

Proof By induction on the sum of the ranks of α and α'. The first case is from the transitivity of $\leq$. For the second case, assume $\phi \leq \phi' \sqsubseteq e$ and $e \Downarrow \lambda x.b$. Notice that $\phi' \sqsubset \lambda x.b$ (since $\lambda x.b \Downarrow \lambda x.b$). Suppose that $< \alpha, \beta > \in \phi$ and $\alpha \sqsubset a$. Since $\phi \leq \phi'$ we know there is $< \alpha', \beta' > \in \phi'$ with $\alpha' \leq \alpha$ and $\beta \leq \beta$! By the induction hypothesis, $\alpha' \sqsubset a$, so by $\phi' \sqsubseteq \lambda x.b$, $\beta' \sqsubseteq b[a/x]$. Again by the induction hypothesis, $\beta \sqsubset b[a/x]$, therefore $\phi \sqsubset \lambda x.b$ as required.

Now consider the quotient case. Suppose $\xi \leq \xi' \sqsubseteq e$ and $e \Downarrow [a]$, thus $\xi' \sqsubset [a]$, so for some $\alpha' \in \xi'$, $\alpha' \sqsubset a$. Since $\xi \leq \xi'$, there is an α in ξ such that $\alpha \leq \alpha$! By the induction hypothesis, since $\alpha \leq \alpha'$ and $\alpha' \sqsubset a$, $\alpha \sqsubset a$. Thus $\xi \sqsubseteq e$.

Qed.

Definition Now we can extend $\sqsubset$ to the term language T_0. We say $e \sqsubset e'$ for e, e' in T_0 if e is $t(\hat{\alpha}_1, \ldots, \hat{\alpha}_n)$ and $\alpha_i \sqsubset a_i \quad 1 \leq i \leq n$ and e' is $t(a_1, \ldots, a_n)$.

Lemma 2.2 *For all e, e', v, v' in T_0 and α in V,*

1. *If $e \Downarrow v$ and $e \sqsubset e'$ then there is a v' such that $e' \Downarrow v'$ and $v \sqsubset v'$.*
2. *If $\alpha \lhd e$ and $e \sqsubset e'$ then $\alpha \sqsubset e'$.*

Proof By induction on the $\lhd$ and $\Downarrow$ relations.

Case e is $ap(f; a)$ and $ap(f; a) \Downarrow \hat{\beta}$ and since $e \sqsubset e'$ we have $e' = ap(f'; a')$ with $f \sqsubset f'$, $a \sqsubset a'$. Also, the rule for application requires that $f \Downarrow \phi$ and for $(\alpha, \beta) \in \phi$, $\alpha \lhd a$. By the induction hypothesis, we know that $\alpha \sqsubset a'$.

We need to find a value v' for $ap(f'; a')$ and show that $\hat{\beta} \sqsubset v'$. By the induction hypothesis we know that $f' \Downarrow u$ for some term u, and $\phi \sqsubset u$. This u can either be a set term or a $\lambda x.b$.

Suppose u is $\hat{\phi}'$; then $\hat{\phi} \sqsubset \hat{\phi}'$, thus $\phi \leq \phi'$ and so there is $< \alpha', \beta' > \in \phi'$ such that $\alpha' \leq \alpha$ and $\beta \leq \beta'$. By Lemma 2.1, $\alpha' \sqsubseteq a'$, so $\alpha' \lhd a'$ and $ap(f'; a') \Downarrow \beta'$ by the rule for application to a set.

Suppose u is $\lambda x.b$, by the definition of $\sqsubset$, $\beta \sqsubset b[a'/x]$. We know that $b[a'/x]$ has a value, say $b[a'/x] \Downarrow v'$, and $\beta \sqsubset v'$. Thus by the λ term application rule, $ap(f'; a') \Downarrow v'$.

Case e is $ap(f; a)$ and $f \Downarrow \lambda x.b$. By the induction hypothesis, $f' \Downarrow \lambda x.b'$ with $\lambda x.b \sqsubset \lambda x.b'$. Since $b[a/x] \sqsubseteq b'[a'/x]$, the the induction hypothesis, there is v' such that $b'[a'/x] \Downarrow v'$ and $v \sqsubset v'$. According to the rule for application, $ap(f'; a') \Downarrow v'$.

The other cases are similar. See Howe [67].

Qed.

Lemma 2.3 *For all $\alpha \in v$, $e \in T_0$, $\alpha \sqsubset e$ iff $\alpha \lhd e$.*

Proof This follows from Lemma 2.2 and the fact that $\sqsubset$ is a subrelation of $\lhd$.

Qed.

Lemma 2.4 *If $\alpha \lhd e[\hat{\beta}/x]$ and $\beta \lhd e'$ then $\alpha \lhd e[e'/x]$.*

Proof By Lemma 2.3, $e[\hat{\beta}/x] \sqsubseteq e[e'/x]$, and by Lemmas 2.2 and 2.3, $\alpha \lhd e[e'/x]$.

Qed.

Now our main theorem is a consequence of Lemma 2.4.

Thm 2.3 *If $\alpha \leq \beta$ and $\beta \lhd e$, then $\alpha \lhd e$.*

2.6.4 Soundness of Nuprl Rules

We now want to show that the core Nuprl rules are sound in Howe's set semantics. First, we define the meaning of sequents. They have the form

$$x_1 : A_1, \ldots, x_n : A_n \vdash t \in T$$

The hypotheses can be dependent in the following way. The type A_2 might depend on x_1, say $A_2(x_1)$. Generally A_{i+1} can depend on $x_1, \ldots, x_i$, say $A_{i+1}(x_1, \ldots, x_i)$, but it has no other free variables. T and t can depend on $x_1, \ldots, x_n$, but there are no other free variables in T and t.

A *closing substitution* for the sequent is a substitution of closed terms $a_1, \ldots, a_n$ for the free variables $x_1, \ldots, x_n$.

Definition The sequent $x_1 : A_1, \ldots, x_n : A_n \vdash t \in T$ is true if $[\![cl(t)]\!]_T \in [\![cl(T)]\!]$ for all closing substitutions cl such that for all i, $1 \leq i \leq n$, $[\![cl(x_i)]\!]_{A_i} \in [\![cl(A_i)]\!]$.

If $S, S_1, \ldots, S_k$ are sequents, then an inference rule

$$\frac{S_1, \ldots, S_k}{S}$$

is *sound* if the conclusion S is true whenever the premises S_i are true.

We will prove the soundness of a few Nuprl rules.

1. Function Introduction

$$\frac{\vdash A \in \mathbb{U}_i \quad x : A \vdash b \in B}{\vdash \ \lambda x.b \in x : A \rightarrow B}$$

The leftmost premise assures that $[\![A]\!] = \gamma$ for some set γ. The other premise assures that for each $\alpha \in \gamma$, $[\![B[\hat{\alpha}/x]\,]\!] = \gamma_\alpha$ for a set γ_α. It also assures that if cl is a closing substitution that assigns x to any α in γ, then

$$[\![b[\hat{\alpha}/x]\,]\!]_{\gamma_\alpha} \in [\![B[\hat{\alpha}/x]\,]\!]$$

so the second premise is true.

We also know that $[\![x : A \rightarrow B]\!]$ is the set of functions ϕ such that for all $\alpha \in \gamma$, $\phi(\alpha) \in \gamma_\alpha$. Let ϕ encode the particular mapping of $\alpha \in \gamma$ to $[\![b\,[\hat{\alpha}/x]\,]\!]_{\gamma_\alpha}$. Then $\phi \lhd \lambda x.b$, hence $[\![\lambda x.b]\!]_{x:A \rightarrow B} \in [\![x : A \rightarrow B]\!]$.

2. Function Elimination

$$\frac{\vdash a \in A \quad \vdash f \in x : A \rightarrow B}{\vdash \ f(a) \in B[a/x]}$$

The leftmost premise is true if $[\![A]\!] = \gamma$ and $[\![a]\!]_A = \alpha \in \gamma$.

The other premise is true if $[\![x : A \rightarrow B]\!]$ is the set of dependent functions ϕ such that for all $\alpha \in \gamma$, $\phi(\alpha) \in [\![B[\hat{\alpha}/x]\,]\!]$, and if $[\![f]\!]_{x:A \rightarrow B} = \phi$ is one of these functions.

To see that the conclusion is true, notice that for any $< \alpha, \beta > \in \phi$, $\beta \in [\![B[\hat{\alpha}/x]\,]\!]$, and since $\alpha \lhd a$ we have

$$[\![B[\hat{\alpha}/x]\,]\!] \lhd B[a/x].$$

Thus, $[\![B[a/x]\,]\!] = [\![B[\hat{\alpha}/x]\,]\!]$, and since $\phi \lhd f$ we have $\beta \lhd f(\hat{\alpha})$ and $f(\hat{\alpha}) \sqsubset f(a)$, so $\beta \lhd f(a)$. This means that $[\![f(a)]\!]_{B(a/x)} \in [\![B[a/x]\,]\!]$ as required.

2.7 Applications to HOL and PVS

Howe's methods suggest a way to enhance the HOL and PVS type theories by allowing untyped lambda terms to be given a set theoretic meaning in the standard set model, enhanced by approximation relations $\alpha \lhd f$ and $\phi \leq \phi'$. Using these approximations we can define a set theoretic sense to polymorphic functions.

The semantics for HOL and PVS suggested here would allow the addition of quotient types with the right computational behavior. It would also allow HOL to introduce records as types of polymorphic functions; see [41, 71].

The forthcoming work of Moran [90] shows how to extend Howe's results to include intersection and union types. He uses results from Aczel [8]. These results might also apply to HOL and PVS.

Chapter 3

Identifier Reference, Theories and Closed Maps

with Stuart Allen

This material is taken from the technical development in Stuart Allen's article, *Abstract Identifiers and Textual Reference* [12].

Human understanding depends on systematic naming of pieces of information. Proper nouns are paramount examples, and in mathematics we use definitions, theorem names, rule names, names for specific assumptions and goals. In automated reasoning where the operation of rewriting one term to another is critical, as in $(a-b)(a+b)$ rewrites to $a^2 - b^2$, we even have systematic names for pieces of expressions, say the "leftmost operand of the product expression."

When we think about collections of definitions and theorems, we are in the habit of naming them systematically and also naming the collection, say as elementary number theory which includes theorems such as the Fundamental Theorem of Arithmetic, the Euclidean algorithm, the Bezout identity, and so forth. Programming languages and theorem proving systems such as PVS, HOL, and Metaprl [62, 61] organize collections into modules or classes or theories. This is a well-accepted practice, but it has its drawbacks.

In this section we want to introduce another set of concepts that we have found useful for organizing formal mathematics and for managing the rich name spaces associated with them.

3.1 Identifiers in Text

Ordinary mathematical text is replete with explicit names for objects, e.g. "prime number," "Fundamental Theorem of Arithmetic," "Theorem 1," "Lemma 1 for Theorem 2," "The corollary of Theorem 1," "Elementary Number Theory," "Howe's proof of Theorem 1," "Auto tactic." We are concerned with mechanisms for determining the exact referent of these names, especially mechanisms that can be used by software systems that support operations on mathematical texts. We are especially interested in inter-textual references, as in citing a book or article or citing definitions and theorems in libraries of formal mathematics.

Programming languages and their runtime environments are a rich source of examples for how names and their references can be managed. There we deal with local variables, global variables, constants, file names, etc. In this setting, identifiers are typically character strings, and their meaning is given by a context.

It is worth noting that the formal digitally manipulated counterparts of these two paradigmatic domains, formal verification and programming, are not independent. In some formal proof development systems (especially "tactic" based systems [52]) program text is included as part of the proof to indicate how it is to be verified, and such verification programs often make explicit reference to proofs as data (as lemmas to be cited, for example). Of course, when a formal argument is about particular program texts, those program texts will be cited in the assertions of the proof. The correctness of claims based on formal texts obviously will often depend upon what those texts contained and how they referenced one another.

The basic management problem for formal texts is how to extend or otherwise alter them without accidentally ruining the bases for claims depending upon them; this becomes an issue of *accounting* for various dependencies between texts. A collection of proof texts might be ruined, for example, if the proof of some lemma were replaced by a new proof using a new axiom not accepted by the reader. Or a program text might be ruined by alteration of a critical subprogram that makes the program as a whole useless.

3.2 Conventional Runtime References

In the case of program text, we can find the "content of" or "referent of" reference values. For example we might use "path" to the referent, perhaps a URL for web lookup, or a file system path, or a RAM address. On the other hand the reference value might be atomic, meaning that it has no particular internal structural relation to any other reference values; maybe it is used as an index in an association list.

Reference values might also be either concrete or abstract; character strings and numbers would be examples of concrete values, whether treated as referentially atomic or not, whereas pointer values as in SIMULA, say, would be abstract*, since they can be used computationally only in comparisons to other pointer values, or to look up their referents, or as whole data passed between computations, and the only way to get a pointer value (other than nil) initially is by allocation of a referent for one. In contrast, concrete values have some "external" identity independent of this or that execution of a program. It would also be possible to base a reference function on a mixed mode, such as values being structured complexes of abstract atoms. Our proposal will be to *equate* identifiers with references to texts and to treat them as both abstract and atomic.

3.3 Constraints on Structure of Individual Texts

Now we consider the structure of texts. We assume there is a class Text(D) of possible texts where D is whatever type of values will play the part of our distinguished identifiers in texts. That is, we parameterize our concept of text by the class of identifiers; quantification over suitable types for D will be key to our treatment of identifiers as abstract.

Before considering plausible candidates for Text(D), let us present the properties of Text(D). We presuppose the concept of identifier occurrences within a text and the concept of replacing those occurrences by other identifiers, as well as the derivative concept "$t/\!:=g$" of uniformly replacing identifiers throughout a text t by applying function g to them.

We assume that there is a precise notion of *identifier occurrences* in texts, for the distinguished sort of identifiers we're trying to introduce. Let the class Occ(t) index the identifier occurrences within $t \in$ Text(D), of which we assume there are finitely many, indexing them independently of both D and which particular identifiers occur in t; we might think of these as places in the text where identifiers can go. We shall presuppose functions id$(t)@(x)$, computing the identifier for each place x in t, and t/f, which assigns an identifier to each place according to an assignment function f, characterized by:

* The notion of abstract reference values seems to have arisen independently with SIMULA [96] and the ancestors of PASCAL, namely EULER [118] and ALGOL-W [117]. Little seems to have been made of it by the authors, perhaps because of its minor significance relative to the other innovations. Conjecture: when one incorporates ram addresses into an Algol60-like semantic method, as with both the above lines of development, they tend to become abstract.

$\forall t$:Text(D). Occ(t) finite

$\forall t$:Text(D'), f:(Occ(t)$\rightarrow D$). $t/f \in$ Text(D)

$\forall t$:Text(D'), f:(Occ(t)$\rightarrow D$). Occ(t/f) $=$ Occ(t)

$\forall t$:Text(D''), g:(Occ(t)$\rightarrow D'$), f:(Occ(t)$\rightarrow D$). $t/g/f = t/f$

$\forall t$:Text(D). id(t)@ $\in$ Occ(t)$\rightarrow D$

$\forall t$:Text(D), f:(Occ(t)$\rightarrow D$). $t = t/f \Leftrightarrow f =$ id(t)@

Text(D) is monotonic in D

$\forall t$:Text(D). $t \in$ Text(Ids(t)), where Ids(t) $\equiv_{\text{def}}$ $\{$ id(t)@(x):D $|$ $x\in$ Occ(t) $\}$

and consequently

$\forall t$:Text(D). $t = t/$id(t)@

$\forall t$:Text(D'), f:(Occ(t)$\rightarrow D$). id(t/f)@ $= f \in$ Occ(t)$\rightarrow D$.

Then $t/{:=}g$ $\equiv_{\text{def}}$ $t/(g \circ$ id(t)@) is the uniform replacement of each identifier x in t by $g(x)$.

We further assume that identity on Text(D) is *effectively decidable* when identity on D is.

3.4 Structured Text and Abstract Syntax

Making identifiers abstract can be seen as the completion of a movement away from concrete syntax, which in the Nuprl research program began with the adoption of *pure structured texts* rather than string text as the fundamental data structure, and which may be seen as well in the more recent widespread adoption of XML on the worldwide web.

In the Nuprl4 systems [68], although all textual constituents are concrete values, the texts are deemed otherwise independent of any string representation. Indeed Nuprl4 was in use long before any conventions for standard string representations were specified (for communication with external processes). How texts are presented for viewing or editing by a user is determined by how the user's session is configured, especially what collections of "display forms" have been activated. These independently loadable display forms can be altered by the user at will, permitting different users to use their preferred notations for the underlying structured texts, or letting a developer proceed with content development without first committing to notations. Further, it is possible to manage notational ambiguity when the browsing or editing tools directly access the underlying unambiguous structured text, and support incremental disambiguation on demand.

The purpose of making content independent of notation was to encourage the use of formal text free of notational vagaries and disputes, simultaneously acknowledging the profound cognitive significance of concrete notations to users and their irrelevance to ordinary formal issues. Concrete notation may be so important to the user that it should not be needlessly dictated by the different needs of others, including authors of formal texts.

The use of structured texts not only liberated the development of formal material from many notational concerns, but actually simplified the programs used to analyze and modify texts both for formal purposes, such as proof verification, and also for utilities for presenting, editing and managing texts. While the simplification of formal textual analysis by programs is dependent on the fact that the text is structured, and therefore cannot be expected to provide models for runtime manipulation of atomic identifiers, *the abstractness of textual structure does give us some insight into how one lives concretely with abstract features of syntax.*

Here are a few practical observations gleaned from experience using abstractly structured text, albeit with concrete identifiers. First, since the user's access to the structured texts is mediated by an interface that realizes (user alterable) notations for the abstract text structures, the user ordinarily does not need to know all the details of the underlying text structure, such as what order the various constituents really occur in, which may differ from the order in which they are displayed. Although the meanings of abstractly structured texts are keyed off of constituent identifiers, like "all" or "implies", exactly which concrete identifiers are so used can typically be ignored by the user because the display of instances typically uses some other suggestive notation not involving those identifiers. For example, the text displayed as "($\exists x{:}\mathbb{Z}.\ x = x) \Rightarrow P$" in the standard Nuprl system setup is actually a complex including the concrete identifiers "implies", "exists", "equal", "int" and "variable", none of which is apparent from the notation or relevant to the understanding or editing of the underlying structured text.

These identifiers do not need to be known to the user because instances of the operators of interest can be used directly; for example, one can find the definition of an operator (or that it is an undefined primitive) directly from a textual instance unmediated, as far as the user need know, by the constituent identifiers. When the user wants to create an instance of an operator the identifier need not be known because it can be built by cut-and-paste from another text, or by clicking in a menu, or by the user entering a command which is bound to the operator. And just as notations for structured text can be controlled by the user, so can these command-bindings, which are similarly independent of the constituent identifiers; for example the command bindings that happened to be used when creating the example text just above were "some", "imp" and "=", rather then "exists", "implies" and "equal".

Further, since program source texts in Nuprl are themselves structured, other text can be included in literals (in quotation) in the source program, and so even the programmer need not be explicitly aware of what the constituent identifiers are in sample texts.

These observations suggest the possibility that the concreteness of identifiers may well be inessential to a person's practical access to structured text when mediated by an appropriate interface that realizes any concrete symbolic "bindings" to abstract text. Below we shall emphasize definite liabilities of concrete identifiers in formal texts, which ought to encourage us to exploit their formal dispensability — by dispensing with them.

3.5 Identifying Abstract Identifiers with References to Texts

We use $D{\to}\text{Text}(D)$ as the type of text reference, making the "content of" or "referent of" function map every identifier, whatever the class of identifiers happens to be, to texts whose identifiers are drawn from that same class. We shall call such functions *closed maps* because every identifier that can occur in a text refers to a text, i.e. there are no dangling pointers.

Closed maps then will serve as inter-referential text collections (considering the choice of domain as part of the map's identity). Claims about (inter-referential) texts will be interpreted with respect to closed maps $\langle D, f\rangle$. Claims that are abstract with regard to identifiers will be about appropriate equivalence classes of closed maps. It will be desirable to make the claims themselves come under management by recording them as texts referring to the texts they are about. Because of our esteem for formal correctness and our hope for free combination of formal texts, we are especially interested in claims that are abstract with regard to identifiers and are based on execution of programs operating on closed maps.

We now elaborate on closed map equivalence and on operations on closed maps that are important for our purposes, why atomic pointers, and why no dangling pointers. We shall indicate equivalence between closed maps by $\langle D, f\rangle \sim \langle D', f'\rangle$ which can be characterized by

$\forall f{:}(D{\to}\text{Text}(D)),\ f'{:}(D'{\to}\text{Text}(D'))$.
$\langle D, f\rangle \sim \langle D', f'\rangle \Leftrightarrow (\exists g{:}(D{\to}D').\ g{:}D{\to}D'$ renames f to f')

where

$g{:}D{\to}D'$ renames f to f' $\equiv_{\text{def}}$ $g{:}D{\to}D'$ invertible & g carries f into f'

g carries f into f' $\equiv_{\text{def}}$ $\forall i{:}D.\ (f(i)/{:=}g) = f'(g(i)) \in \text{Text}(D')$

The notion of "carrying" $f \in D{\rightarrow}\text{Text}(D)$ into $f' \in D'{\rightarrow}\text{Text}(D')$ is that of embedding the texts and intertextual references modulo consistent change of identifiers, but further allowing for the identification ("collapsing") of some identifiers, which can be useful independently of just renaming. Renaming is carrying without collapsing distinct identifiers by requiring the identifier translation to realize a one-to-one (i.e. invertible) correspondence between the identifier types, hence when $D = D'$ renaming is simply permuting all the identifiers.

It will be convenient to avail ourselves of a vernacular of "objects" and "pointers" in discussing closed maps. The object-vernacular works by pretending that the current closed map relation between identifiers and texts is mediated by some "objects" which have the identifiers as names or pointers and the texts as content; these are concepts we are all accustomed to using informally, and which may help to indicate the practical purposes of this proposal. This pretense allows us to think of changing the contents of an object or its name independently. Sometimes it will be convenient to identify objects with identifiers, and talk about changing contents of, deleting, creating or copying objects. Other times it will be convenient to consider the identifiers as arbitrarily assigned, and freely reassignable, object names. But always our vernacular here shall allude to facts about closed maps *per se*.

Returning to closed maps, it may help to be precise about a few concepts.

3.6 Closed Submaps and Indirect Reference.

The concept of closed map equivalence $M \sim M'$ (along with the cognate concepts of "renaming" the identifiers of a map and "carrying" one map into another) was introduced because respecting this equivalence is the principal stance towards text collections motivating closed maps. Here are some other useful concepts made explicit, along with some glosses in the object/pointer vernacular, pertaining to indirect reference and to closed submaps.

The concept of a text referring directly to an object is obvious but doesn't require the reference relation as an explicit parameter, since it is just the occurrence of a pointer value in the text:

t:Text(D) refers directly to $j \ \equiv_{\text{def}}\ \exists p$:Occ($t$). $j = \text{id}(t)@(p) \in D$

Ids(t) $= \{i{:}D|\ t$:Text(D) refers directly to i $\}$

Possibly indirect reference, of course, depends on the map:

t:Text(D) refers to j via $f \ \equiv_{\text{def}}\ \exists n{:}\mathbb{N}$. t:Text(D) reaches j via f within n steps

which means that the pointers are followed through the map. Composing this relation with the map itself naturally extends it to inter-object reference, for example i refers (perhaps indirectly) to j when $f(i)$:Text(D) refers to j via f. We extend this to a reflexive relation between objects:

$i{:}D$ reaches j via $f \ \equiv_{\text{def}}\ j = i \in D \lor f(i)$:Text($D$) refers to j via f

A (closed) submap of a closed map is simply a restriction of the identifier class, *given* that the map remains closed. Here is a characterization of the closed submap relation $M \subseteq M'$.

$\forall f{:}(D{\rightarrow}\text{Text}(D)),\ f'{:}(D'{\rightarrow}\text{Text}(D'))$.
$\langle D, f\rangle \subseteq \langle D', f'\rangle \Leftrightarrow (D \subseteq D')$ & $(\forall i{:}D.\ (\text{Ids}(f'(i)) \subseteq D)$ & $f(i) = f'(i) \in \text{Text}(D))$

Note that there is no renaming involved in this simple notion of submap. Now let us consider some important ways of finding submaps. Although we cannot restrict a closed map to just any subclass X of its identifiers, we can restrict it to the smallest submap comprising X, which we shall call "contracting around" X:

$\forall f{:}(D{\rightarrow}\text{Text}(D))$.
$(X \subseteq D) \Rightarrow$ Contract $\langle D, f\rangle$ around $X = \langle\{j{:}D|\ \exists i{:}X.\ i{:}D$ reaches j via f $\}, f\rangle$

This fundamental operation allows us to select "coherent" subparts of a closed map by stipulating what part we are primarily interested in and retaining everything that part depends on. A cognate operation is "focusing" on the part relevant to identifier subclass X, which is contracting around all the identifiers that can reach X:

$\forall f{:}(D{\rightarrow}\text{Text}(D))$.
$(X \subseteq D) \Rightarrow$ Focus $\langle D, f\rangle$ on $X =$ Contract $\langle D, f\rangle$ around $\{i{:}D |\ \exists y{:}X.\ i{:}D$ reaches y via $f\ \}$

In the other direction, to discard several objects X requires discarding all that point to them as well, i.e. contracting around the type of identifiers neither in X nor pointing even indirectly to any of X:

$\forall f{:}(D{\rightarrow}\text{Text}(D))$.
$(X \subseteq D) \Rightarrow$
Delete X from $\langle D, f\rangle =$ Contract $\langle D, f\rangle$ around $\{i{:}D |\ \neg(\exists y{:}X.\ i{:}D$ reaches y via $f)\ \}$

The significance of a submap is not only that it is a natural unit of coherency for an object collection, but that when claims are "localized" they are about texts only of a particular submap, and if the localized claim is recorded as a text in the map itself, then any submap or supermap containing that record preserves the claim.

Chapter 4

Event System Specifications in Type Theory

with Mark Bickford

4.1 Introduction

One of the goals of *formal methods research* is to increase the capacity of system designers, programmers, and engineers to create highly reliable systems that perform well and evolve economically to ever-higher reliability and better performance.

Currently formal methods are too expensive for widespread use, and they are thus confined to select components of critical systems. One way to reduce the cost is to fully automate the tools, as in the case of type checking, extended static checking, model checking, decision procedures, and fully automatic verification. This approach is being vigorously pursued [35, 103, 35, 114, 13, 36, 46, 23].

Another way to reduce the cost is to provide better interactive tools and extensive libraries of formal knowledge to experts so that they can be more effective. We discovered that in our work on distributed system verification [41], we accumulated sufficient knowledge that *we could participate in the protocol design process at the speed of the designers and programmers,* and provide formal guarantees, formally justified designs, and formal explanations.

This section will illustrate how we formalized in type theory the knowledge base needed to reason about distributed systems and specific protocols. Our early work relied on a formalization of a version of Lynch's IO Automata [80]. Our current work uses the related idea of a *message automaton*[23]. Our specifications rely on the notion of an event system, which is an abstraction from the notion of a computation of a system of interacting message automata.

There isn't time in these lectures to examine a serious verification, or to illustrate protocol synthesis from constructive proofs, but we can present the flavor of the mathematics and consider how PVS and Nuprl can collaborate in this arena. So our approach is to examine the basic mathematics of event systems and its formalization in type theory.

4.2 Event Systems

An *event system* is a mathematical structure designed to express the key features of a distributed computing system at the level of abstraction appropriate for specifying interactive behavior without regard to particular machines or protocols. It allows constructive expression of the concepts used by Lamport [75, 76] and Winskel [115, 116] to explain distributed systems, and the concepts used by Birman [27, 28], van Renesse [29], Dolev [13] and their colleagues to design them, e.g. Isis [29], Transis [13], Horus [112], and Ensemble[26, 60].

While an event system is a bit more elaborate than mathematical structures such as vector spaces, the complexity is comparable to the notion of a polynomial algebra. There are only seven axioms, and they relate these notions.

There is a type E of *events;* each event e is associated with an address or process which localizes the event. We say that the event happens at a locus, Loc, although this need not be a point in space. We use the type Loc for the localizers, i. Associated with the locus of events are observable objects or values named by identifiers, Id. We use $x, y, z, \ldots$ to name them. Each value at a locus has a type, $T(i, x)$.

The communication topology of an event system is given by connecting locations by explicit links, l, in the type Lnk. Each $l \in Lnk$ has a source, src, and destination, dst.

4.2.1 Example — Two-Phase Handshake

Here is an illustration of the concepts on an extremely simple protocol — a two-phase handshake — in which a sending process S transmits messages on a link l_1 to a receiver R in such a manner that the previous message is acknowledged by R on link l_2 before the next one is sent by S.

We can define the events whose locus is S and R as

$$E_S == \{e : E \mid loc(e) = S\}$$

$$E_R == \{e : E \mid loc(e) = R\}$$

where E is the type of all events and $loc(e)$ computes the process identifier or the locus of the event. The process identifiers or addresses are from the discrete type Loc, i.e. $S, R \in Loc$.

Some events at S send messages on the link l_1 between S and R. We say that the source of l_1 is S and the destination is R, $src(l_1) = S, dst(l_1) = R$; and $src(l_2) = R, dst(l_2) = S$.

The events at S that send on l_1 can be defined as

$$Snd_{S,l_1} = \{e : E_S \mid sends(l_1, e) \neq nil\}.$$

The function $sends(l, e)$ takes a link l and event e and produces a list of tagged messages that are sent on l when the event e occurs.

$$Rcv_{S,l_2} = \{e : E_s \mid kind(e) \text{ is receive on } l_2\}$$

Following Lamport, we define a *causal ordering* relation on events, $e < e'$, as follows. First, we postulate a total ordering on events at any locus defined using an immediate predecessor function, $pred$. We then define $e \;\; L \;\; e'$ as

$$e \;\; L \;\; e' \text{ iff} \quad \begin{array}{l} e' \text{ is not the first event at its locus and } e = pred(e'), \\ \text{or } e' \text{ is a receive and } e = sender(e'). \end{array}$$

Now define $e < e'$ as the transitive closure of L.

We can now write the top level specification for S

$$\forall e_1, e_2 : Snd_{S_1 l_1}. \quad \exists r : Rcv_{S,l_2}. \quad e_1 < e_2 \Rightarrow (e_1 < r < e_2),$$

and for R we stipulate

$$\forall e_1, e_2 : Snd_{R,l_2}. \quad \exists r_1 r_2 : Rcv_{R,l_1}. \quad e_1 < e_2 \Rightarrow (r_1 < e_1 < r_2 < e_2).$$

4.2.2 Types and Axioms

There are two kinds of events: either a *local action,* a in Act; or a *receive action*. The receive actions happen on a link, they are tagged by elements of *Tag,* and they have values associated with them — the content of received messages. Every event can send a list of tagged messages on a link, so a send is not a separate kind of event.

The types E, Loc, Id, Lnk, Act, and *Tag* must all be *discrete*, that is, equality on them is decidable. Let $\mathbb{D} == \{T : type_i \mid \forall x, y : T. \quad Decidable(x = y \text{ in } T)\}$.

We need functions that type the various components and supply information.

$T : Loc \to Id \to Type_i$

$V : \mathit{Knd}(Lnk, Tag, A) \to Loc \to Type_i$

Info: $E \to (Loc \times Act) + (Lnk \times Tag \times E)$

$src, dst : Lnk \to Loc$

$pred : E \to E?$

We axiomatize event systems in terms of a predecessor function on E, $pred(e)$, which returns the element of the unit type if e is the first event; otherwise, it returns the immediately prior event. We also use a function, val:

$$e : E \to V(kind(\mathit{info}, e), loc(e)),$$

and two temporal operators, $when$, *after*:

$$x : Id \to e : E \to T(loc(e), x).$$

Here are the axioms stated formally and informally.

Ax 1 *For every event e which sends a list of messages on link l, we can find an event e' at the destination of l at which all messages sent are received.*

$$\begin{array}{l} \forall e : E. \quad \forall l : Lnk. \quad \exists e' : E. \quad \forall e'' : E. \\ \quad receive(e'') \Rightarrow e = sender(e'') \Rightarrow link(e'') = l \Rightarrow \\ \quad (e' = e'' \vee e'' < e') \wedge loc(e') = dst(l) \end{array}$$

Ax 2 *The predecessor function at each locus is one-to-one.*

$$\forall e, e' : E. \quad loc(e) = loc(e') \Rightarrow (pred(e) = pred(e') \Rightarrow e = e')$$

Ax 3 *The causal order $<$ is strongly well-founded.*

$$\forall e : E. \quad \{e' : E \mid e' < e\} \textit{ is finite}.$$

Ax 4 *If e is not the first event at a locus, then its predecessor is at the same location.*

$$\forall e : E. \quad \neg \mathit{first}(e) \Rightarrow loc(pred(e)) = loc(e).$$

Ax 5 *If e is a receive event, then the locus of the sender is the source of the link of e.*

$$\forall e : E. \quad receive(e) \Rightarrow loc(sender(e)) = src(link(e)).$$

Note, we can compute the link of any receive event.

Ax 6 *Messages on a link are received in the order sent.*

$$\begin{aligned}\forall e, e' : E. \quad & (rcv(e) \Rightarrow rcv(e')) \Rightarrow \\ & \quad (link(e) = link(e')) \Rightarrow \\ & \quad\quad (sender(e) < sender(e')) \Rightarrow e < e'.\end{aligned}$$

Ax 7 *By convention, for any event except the first, to say that an observable x has a value v when event e happens at the locus of e is to say that x after the predecessor of e is v.*

$$\forall e : E. \quad \neg \mathit{first}(e) \Rightarrow \forall x : Id. \quad (x \ \mathit{when} \ e = x \ \mathit{after} \ \mathit{pred}(e)).$$

4.2.3 Executions of Distributed Systems

Event systems are relevant to distributed computing because for a variety of definitions of a distributed system, every *computation* (or *run* or *execution*) produces an event system in a natural way. We illustrate this for a rather general definition, and we have done it as well for models in which the communication topology is a graph whose nodes are IO Automata or message automata [23].

We consider a distributed system to be given by a graph whose nodes are Loc and whose edges are Lnk connected in the obvious way. At every node there is a *state space* which is a record $\{x_1 : A_1; \ldots ; x_n : A_n\}$ where the $x_i \in Id$; the elements are states.

A computation is a sequence of states indexed by location and time, say $s(i, t)$, together with lists of tagged messages on every link, say $m(l, t)$. Time is discrete; we take it to be indexed by the natural numbers $\mathbb{N}$. Additionally, we can name the action taken at location i and time t, $a(i, t)$.

We will assume that computations are *fair-fifo*. That means that

1. Only the process at i can send messages on links originating at i.
2. A receive action at i must be on a link whose destination is i and whose message is at the head of the queue on that link.
3. There can be null actions that leave a state unchanged between t and $t + 1$.
4. Every queue is examined infinitely often, and if it is nonempty, a message is delivered.
5. The precondition of every local action is examined infinitely often and if true the action is taken.

Given a fair-fifo execution w, we can define an event system from it, denoted $Ev(w)$. The *events* are the points $< i, t >$ at which a non-null action occurred in w.

4.2.4 Deriving and Verifying the Two-Phase Handshake Protocol

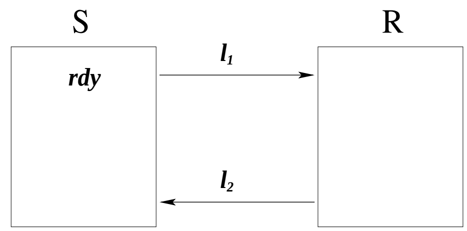
We would like to derive the two-phase handshake protocol by successively refining the specifications into conditions on the processes S and R. At some point we see how to write processes with these properties or we see that some protocol has them. We illustrate this method next.

One way to achieve an alternation of sends and receives at S is to introduce another observable, say rdy, which is a boolean. We then stipulate that S sends only when $rdy = true$ and that when the sends occur rdy is set to *false*. Finally we say that only a receive event at S can cause an internal action that sets rdy to $true$. We also postulate that when $rdy = true$, S will eventually send.

Lemma 4.1 *For any two processes S, R, and links*

$$l_1, l_2, \ src(l_1) = S \ \& \ dst(l_1) = R, \ src(l_2) = R \ \& \ dst(l_2) = S, \ \& \ rdy \in Id,$$

we can realize an event system with the property that initially $rdy = true$ at S.



Let us call these systems $ES(S, R, l_1, l_2, rdy)$, and the realizer ES_0 for short.

Lemma 4.2 *We can find a realizer that extends ES_0 such that for any non-empty type T*

$$\begin{array}{l} \forall e : E_S. \quad (rdy\ \textit{when}\ e = true) \Rightarrow \\ \qquad \exists e' : E_S. \quad \exists v : T. \quad (e < e' \wedge rdy\ \textit{after}\ e' = \textit{false} \wedge \\ \qquad\qquad\qquad sends(e', l_1) = [< l_1, val, v >]) \textit{ or } (rdy\ \textit{when}\ e = \textit{false}). \end{array}$$

Proof We specify an action which has a precondition that $rdy = true$. In this case, it takes the local action of setting rdy to *false* and sending a message to S tagged by val whose content is an element of T. The event e' is the action at this point in time.

Qed.

Call the realizers from Lemma 4.2 ES_1.

Lemma 4.3 *We can find a realizer that extends ES_1 such that*

$$\forall e : E_S. \quad sends(e, l_1) \neq nil \Rightarrow rdy\ \textit{after}\ e = \textit{false}.$$

Proof We constrain the realizer of Lemma 4.3 to satisfy the condition that there are no actions that send on l_1 with tag val except for the one specified in Lemma 4.3.

Qed.

Call the realizers of Lemma 4.3, ES_2.

Lemma 4.4 *We can find a realizer that extends ES_2 such that*

$$\forall e : E_S. \quad rcv(e, l_2) \Rightarrow rdy\ \textit{after}\ e = true.$$

Proof We add to the process at S a response to any receive action on l_2, namely this receive will set rdy to $true$.

Qed.

Call the realizer of Lemma 4.4 ES_3.

Lemma 4.5 *We can find realizers that extend ES_3 such that*

$$\forall e : E_S. \quad rcv(e, l_2) \Rightarrow \exists e' : E_S. \quad e < e' \wedge sends(e, l_1) \neq nil$$

Proof The system ES_4 will guarantee this provided rdy is not set to *false* before the internal action can detect that it is $true$. So we enforce a "frame condition" that restricts the event system, so that only the receive on l_1 and the reset of rdy to *false* can affect rdy. We also stipulate that execution of local actions is also fair, they are examined infinitely often, and if their precondition is $true$, they are executed.

Qed.

Call the realizer of Lemma 4.5 ES_4.

Thm 4.1 *Any realizer extending ES_4 satisfies*

$$\forall e_1, e_2 : Snd_{S,l_1}. \quad (e_1 < e_2 \Rightarrow \exists r : Rcv_{S,l_2}. \quad e_1 < r < e_2)$$

Proof Let e_1, e_2 be send events at S on link l_1. Suppose $e_1 < e_2$. By Lemma 4.3, rdy after $e_1 =$ *false*. The only event that can set rdy to $true$ is a receive event. The only way $e_1 < e_2$ is possible is that rdy when $e_2 = true$. Thus before e_2 and after e_1 there must be a receive event.

Qed.

We can prove similar facts about process R and then by combining the event systems, into ES', we can prove that any extension of ES' satisfies the combined specification.

Our proof of the specification was achieved by implicitly building a protocol. We used a very high-level description instead of program code. We could also proceed by writing code, say using IO Automata, and showing that the executions define an event system ES' all of whose extensions satisfy the specification. In addition, we can arrange to extract message automata from constructive proofs that a specification is achievable. There is no time to explore this topic here, but see the article *A Logic of Events* [23].

Regardless of the method of building the protocols, we think that event systems provide a natural means of stating specifications and describing distributed systems.

4.3 Event System Properties

The simple event systems of Section 4.2 can be enhanced with additional defined operations. Here is a more pragmatic system that can be derived from the simple one.

4.3.1 Extending Event Systems

An *event system* is graph, kind, message structures

$$(Loc, Lnk, src, dst),\ (Kind, Lnk, isrcv, lnk, tag, rcv, act),\ \text{and}\ (Msg, M, lnk, tag, mval, <\_,\_,\_>)$$

and an event structure

$$(E, T, V, loc, kind, val, \textbf{when}, \textbf{after}, \textbf{initially}, sends, <_{loc}, \mapsto, \prec) \text{ where}$$

$E, Loc, Lnk, Kind : \mathbb{U}$	events,locations,links,kinds,messages
$T : Lbl \to Loc \to \mathbb{U}$	T(x,i) is type of x at i
$V : Kind \to Loc \to \mathbb{U}$	V(k,i) is type of value k(v) at i
$loc : E \to Loc$	loc(e) = location of e
$kind : E \to Kind$	kind(e) = kind of e
$val : e : E \to V(kind(e), loc(e))$	val(e) = value of e
$\textbf{when} : x : Lbl \to e : E \to T(x, loc(e))$	value of x when e occurs

after $: x : Lbl \rightarrow e : E \rightarrow T(x, loc(e))$	value of x after e occurs
initially $: x : Lbl \rightarrow i : Loc \rightarrow T(x, i)$	initial value of x at i
$sends : E \rightarrow Msg\ List$	sends(e) = messages sent by e
$<_{loc}: E \rightarrow E \rightarrow \mathbb{P}$	$e\ <_{loc}\ e' \iff e$ locally before e'
$\mapsto: E \rightarrow \mathbb{N} \rightarrow E \rightarrow \mathbb{P}$	$e, n\ \mapsto\ e' \iff e'$ is rcv nth $sends(e)$
$\prec: E \rightarrow E \rightarrow \mathbb{P}$	$e \prec e' \iff e$ causally before e'

Using these primitives we define:

$$
\begin{aligned}
msg(e) &= \ < lnk(kind(e)), tag(kind(e)), val(e) > \\
&\qquad \text{only defined when } isrcv(kind(e)) \\
e = pred(e') &= e\ <_{loc}\ e' \ \wedge\ \forall e'' : E.\ \neg(e\ <_{loc}\ e'' \ \wedge\ e''\ <_{loc}\ e') \\
first(e) &= \forall e'.\ \neg e'\ <_{loc}\ e \\
e\ \mapsto\ e' &= \exists n : \mathbb{N}.\ (e, n\ \mapsto\ e') \\
sends(l, e) &= filter((\lambda m. lnk(m) = l), sends(e)) \\
x\ \Delta\ e &= (x\ \textbf{after}\ e \neq x\ \textbf{when}\ e)
\end{aligned}
$$

An event system will also satisfy the following axioms

1	$\forall e, e' : E.\ e = pred(e') \ \Rightarrow\ \forall x : Lbl.\ x\ \textbf{after}\ e = x\ \textbf{when}\ e'$
2	$\forall e : E.\ first(e) \ \Rightarrow\ \forall x : Lbl.\ x\ \textbf{when}\ e = x\ \textbf{initially}\ loc(e)$
3	$\forall e, e' : E.\ loc(e) = loc(e') \ \Rightarrow\ e\ <_{loc}\ e' \vee e = e' \vee e'\ <_{loc}\ e$
4	$<_{loc}$ is a decidable, well-founded, transitive, anti-reflexive ordering
5	$\forall e : E.\ isrcv(kind(e)) \ \Rightarrow\ loc(e) = dst(lnk(kind(e)))$
6	$\forall e : E.\ \forall ms : Msg.\ ms \in sends(e) \ \Rightarrow\ loc(e) = src(lnk(m))$
7	$\forall e, e' : E.\ e\ \mapsto\ e' \ \Rightarrow\ isrcv(kind(e'))$
8	$\forall e' : E.\ isrcv(kind(e)) \ \Rightarrow\ \exists e : E.\ \exists n : \mathbb{N}.$ $e, n\ \mapsto\ e' \ \wedge\ msg(e') = nth(n, sends(l, e))$
9	$\forall e_1, e_2, e' : E.\ \forall n_1, n_2 : \mathbb{N}.\ (e_1, n_1\ \mapsto\ e' \ \wedge\ e_2, n_2\ \mapsto\ e') \ \Rightarrow$ $e_1 = e_2 \ \wedge\ n_1 = n_2$
10	$\forall e, e' : E.\ e\ \mapsto\ e' \ \Rightarrow\ msg(e') \in sends(e)$
11	$\forall e_1, e_2, e'_1, e'_2 : E.\ e_1\ \mapsto\ e'_1 \ \wedge\ e_2\ \mapsto\ e'_2 \ \Rightarrow$ $e_1\ <_{loc}\ e_2 \ \wedge\ loc(e'_1) = loc(e'_2) \ \Rightarrow\ e'_1\ <_{loc}\ e'_2$
12	$\forall e, e'_1, e'_2 : E.\ \forall n_1, n_2 : \mathbb{N}.\ e, n_1\ \mapsto\ e'_1 \ \wedge\ e, n_2\ \mapsto\ e'_2 \ \Rightarrow$ $n_1 < n_2 \ \wedge\ loc(e'_1) = loc(e'_2) \ \Rightarrow\ e'_1\ <_{loc}\ e'_2$
13	$\forall e, e' : E.\ e\ <_{loc}\ e' \ \Rightarrow\ e \prec e'$
14	$\forall e, e' : E.\ e\ \mapsto\ e' \ \Rightarrow\ e \prec e'$
15	$\prec$ is a decidable, well-founded, transitive, anti-reflexive ordering
16	$\forall e : E.\ \forall l : Lnk.\ \forall n : \mathbb{N}.\ 0 < n \leq \|sends(l, e)\| \ \Rightarrow$ $\exists e' : E.\ e, n\ \mapsto\ e' \ \wedge\ lnk(kind(e')) = l$
17	$\forall e : E.\ \forall l : Lnk.\ \forall tg : Lbl.\ \forall v : M(l, tg).$

$$< l, tg, v >\in sends(e) \;\Rightarrow\; \exists e'@ = rcv_l(tg)(v).\; e \prec e'$$

We make some shorthand notations:

$$
\begin{aligned}
&\forall e@i.\ \phi \ \equiv \\
&\quad \forall e : E.\ loc(e) = i \ \Rightarrow\ \phi \\
&\forall e@i = pred(e').\ \phi \ \equiv \\
&\quad \forall e, e' : E.\ loc(e) = i \ \wedge\ e = pred(e') \ \Rightarrow\ \phi \\
&\forall e@i = k(v).\ \phi \ \equiv \\
&\quad \forall e : E.\ \forall v : V(k,i).\ loc(e) = i \ \wedge\ kind(e) = k \ \wedge\ val(e) = v \ \Rightarrow\ \phi \\
&\forall e = k(v).\ \phi \ \equiv \\
&\quad \forall e : E.\ \forall i : Loc.\ \forall v : V(k,i).\ loc(e) = i \ \wedge\ kind(e) = k \ \wedge\ val(e) = v \ \Rightarrow\ \phi \\
&\exists e@i.\ \phi \ \equiv \\
&\quad \exists e : E.\ loc(e) = i \ \wedge\ \phi \\
&\exists e@i = k(v).\ \phi \ \equiv \\
&\quad \exists e : E.\ \exists v : V(k,i).\ loc(e) = i \ \wedge\ kind(e) = k \ \wedge\ val(e) = v \ \wedge\ \phi \\
&\exists e = k(v).\ \phi \ \equiv \\
&\quad \exists e : E.\ \exists i : Loc.\ \exists v : V(k,i).\ loc(e) = i \ \wedge\ kind(e) = k \ \wedge\ val(e) = v \ \wedge\ \phi \\
&\exists e' >_{loc} e.\ \phi \ \equiv \\
&\quad \exists e' : E.\ e \ <_{loc}\ e' \wedge \phi \\
&\exists e' <_{loc} e.\ \phi \ \equiv \\
&\quad \exists e' : E.\ e' \ <_{loc}\ e \wedge \phi \\
&\exists e' \geq_{loc} e.\ \phi \ \equiv \\
&\quad \exists e' : E.\ e \ \leq_{loc}\ e' \wedge \phi \\
&\exists e' \leq_{loc} e.\ \phi \ \equiv \\
&\quad \exists e' : E.\ e' \ \leq_{loc}\ e \wedge \phi
\end{aligned}
$$

4.3.2 Consequences of the axioms

We state as lemmas some properties that follow from the axioms.

$$AntiReflexive(<_{loc}) \tag{4.1}$$

$$AntiReflexive(\prec) \tag{4.2}$$

$$\forall e, e' : E.\ e \ <_{loc}\ e' \ \Leftrightarrow\ \neg first(e') \ \wedge\ e \ \leq_{loc}\ pred(e') \tag{4.3}$$

$$\forall e, e' : E.\ e \ <_{loc}\ e' \ \wedge\ \forall e_1 : E.\ \neg(e \ <_{loc}\ e_1 \ <_{loc}\ e') \ \Rightarrow \quad e = pred(e') \tag{4.4}$$

$$\forall e, e' : E.\ Decidable(e \ <_{loc}\ e') \tag{4.5}$$

$$\forall e, e' : E.\ Decidable(e \ \prec\ e') \tag{4.6}$$

$$\forall e : E.\ \forall l : Lnk.\ \forall tg : Lbl.\ \forall v : M(l, tg).\quad msg(l, tg, v) \in sends(l, e) \ \Rightarrow\ \exists e' = rcv_l(tg)(v).\ e \prec e' \tag{4.7}$$

proofs: Lemmas 4.1 and 4.2 follow from the general fact that

$$WellFounded(Rel) \ \Rightarrow\ AntiReflexive(Rel)$$

Suppose $e <_{loc} e'$. From the axioms we conclude $\neg first(e')$, and from the axioms we conclude

$$pred(e') <_{loc} e' \wedge \forall e'' : E. \neg(pred(e') <_{loc} e'' <_{loc} e')$$

So $\neg(pred(e') <_{loc} e)$ and hence, $e \leq_{loc} pred(e')$, which proves lemma 4.3. If we also have $\forall e_1 : E. \neg(e <_{loc} e_1 <_{loc} e')$ then $\neg e <_{loc} pred(e')$, so $e = pred(e')$, which proves lemma 4.4.

We may now prove lemma 4.5 by induction. By lemma 4.3 it's enough to decide $\neg first(e') \wedge e \leq_{loc} pred(e')$, but this is decidable by the induction hypothesis, and the decidability of equality in E. The proof of lemma 4.6 is similar.

If $msg(l, tg, v) \in sends(l, e)$ then for some $n < \|sends(l, e)\|$, $msg(l, tg, v) = nth(n, sends(l, e))$. By the axioms there is an e' such that

$$isrcv_l(kind(e')) \wedge sender(e') = e \wedge index(e') = n$$

So,

$$val(e') = mval(msg(l, tg, v)) \wedge tg = mtag(msg(l, tg, v))$$

That implies that $e' = rcv_l(tg)(v)$ and since $e = sender(e')$ we have $e \prec e'$. This proves lemma 4.7.

4.3.3 Local histories

An event system is a rich enough structure that we can define various "history" operators that list or count previous events having certain properties. Because we can define operators like these we do not need to add "history variables" to the states in order to write specifications and and prove them.

The basic history operator lists all the prior events at a location.

Definition

$$\begin{array}{rcl}
\mathbf{before}(e) & = & \mathbf{if}\ first(e)\ \mathbf{then}\ []\ \mathbf{else}\ pred(e) :: \mathbf{before}(pred(e)) \\
\mathbf{between}(e_1, e_2) & = & [e' \in \mathbf{before}(e_2) \mid e_1 <_{loc} e'] \\
rcvs(l, \mathbf{before}(e)) & = & [e' \in \mathbf{before}(e) \mid isrcv(kind(e')) \wedge lnk(kind(e')) = l] \\
rcvs(l, tg, \mathbf{before}(e)) & = & [e' \in rcvs(l, \mathbf{before}(e)) \mid tag(kind(e')) = tg] \\
snds(l, \mathbf{before}(e)) & = & concatenate([sends(l, e') \mid e' \in \mathbf{before}(e)]) \\
snds(l, \mathbf{before}(e, n)) & = & snds(l, \mathbf{before}(e))\ \mathbf{append}\ firstn(n-1, sends(l, e)) \\
snds(l, tg, \mathbf{before}(e)) & = & [m \in snds(l, \mathbf{before}(e)) \mid tag(m) = tg]
\end{array}$$

Using these operators we can state the following important lemma.

Lemma Fifo

$$\begin{array}{l}
\forall e' : E. \forall l : Lnk. isrcv_l(e') \Rightarrow \\
snds(l, \mathbf{before}(sender(e'), index(e'))) = [emsg(e) \mid e \in rcvs(l, \mathbf{before}(e'))]
\end{array}$$

proof: The proof is by induction on $<_{loc}$. Suppose $isrcv_l(e')$. If

$$snds(l, \mathbf{before}(sender(e'), index(e'))) = \mathbf{nil}$$

then $rcvs(l, \mathbf{before}(e'))$ must also be **nil** because, if $e <_{loc} e'$ is a receive on l then $\langle sender(e), index(e)\rangle <_{loc} \langle sender(e'), index(e')\rangle$ which makes $snds(l, \mathbf{before}(sender(e'), index(e')))$ non empty.

Otherwise, let

$$ms = last(snds(l, \mathbf{before}(sender(e'), index(e'))))$$

then for some $\langle e, n\rangle <_{loc} \langle sender(e'), index(e')\rangle$,

$$snds(l, \textbf{before}(sender(e'), index(e'))) = snds(l, \textbf{before}(e, n))\ \textbf{append}\ [ms]$$

By the axioms, $\exists e'' : E.\ isrcv_l(kind(e'')) \wedge sender(e'') = e \wedge index(e'') = n$, $e'' <_{loc} e'$, so by induction,

$$snds(l, \textbf{before}(e, n)) = [emsg(e) \mid e \in rcvs(l, \textbf{before}(e''))]$$

If there were an e''' with $isrcv_l(e''')$ and $e'' <_{loc} e''' <_{loc} e'$ then by axiom (**??**)

$$\langle e, n\rangle <_{loc} \langle sender(e'''), index(e''')\rangle <_{loc} \langle sender(e'), index(e')\rangle$$

So, $nth(index(e'''), sends(l, sender(e''')))$ would come after ms in $snds(l, \textbf{before}(sender(e'), index(e')))$ contradicting the choice of ms as the last of the list. Thus $rcvs(l, \textbf{before}(e')) = rcvs(l, \textbf{before}(e''))$ **append** $[e'']$ and since, by axiom (**??**), $ms = emsg(e'')$, we have

$$snds(l, \textbf{before}(sender(e'), index(e'))) = [emsg(e) \mid e \in rcvs(l, \textbf{before}(e'))]$$

$\square$

Corollary

$$\begin{array}{l} kind(e') = rcv_l(tg) \Rightarrow \\ \|snds(l, tg, \textbf{before}(sender(e'), index(e')))\| = \|rcvs(l, tg, \textbf{before}(e'))\| \end{array}$$

$\square$

4.3.4 Event system shorthands

We make some shorthand notations:

$$\begin{array}{l}
\forall e@i.\ \phi \equiv \\
\quad \forall e : E.\ loc(e) = i \Rightarrow \phi \\
\forall e@i = pred(e').\ \phi \equiv \\
\quad \forall e, e' : E.\ loc(e) = i \wedge e = pred(e') \Rightarrow \phi \\
\forall e@i = k(v).\ \phi \equiv \\
\quad \forall e : E.\ \forall v : V(i,k).\ loc(e) = i \wedge kind(e) = k \wedge val(e) = v \Rightarrow \phi \\
\forall e = k(v).\ \phi \equiv \\
\quad \forall e : E.\ \forall i : Loc.\ \forall v : V(i,k).\ loc(e) = i \wedge kind(e) = k \wedge val(e) = v \Rightarrow \phi \\
\exists e@i.\ \phi \equiv \\
\quad \exists e : E.\ loc(e) = i \wedge \phi \\
\exists e@i = k(v).\ \phi \equiv \\
\quad \exists e : E.\ \exists v : V(i,k).\ loc(e) = i \wedge kind(e) = k \wedge val(e) = v \wedge \phi \\
\exists e = k(v).\ \phi \equiv \\
\quad \exists e : E.\ \exists i : Loc.\ \exists v : V(i,k).\ loc(e) = i \wedge kind(e) = k \wedge val(e) = v \wedge \phi \\
\exists e' >_{loc} e.\ \phi \equiv \\
\quad \exists e' : E.\ e <_{loc} e' \wedge \phi \\
\exists e' <_{loc} e.\ \phi \equiv \\
\quad \exists e' : E.\ e' <_{loc} e \wedge \phi
\end{array}$$

$$
\begin{aligned}
&\exists e' \geq_{loc} e.\, \phi \ \equiv \\
&\qquad \exists e' : E.\ e \ \leq_{loc} \ e' \wedge \ \phi \\
&\exists e' \leq_{loc} e.\, \phi \ \equiv \\
&\qquad \exists e' : E.\ e' \ \leq_{loc} \ e \wedge \ \phi
\end{aligned}
$$

Appendix A

Derivation of a Fast Integer Square Root Algorithm

by Christoph Kreitz

A.1 Deriving a Linear Algorithm

The standard approach to proving $\forall n\, \exists r\; r^2{\le}n \wedge n{<}(r{+}1)^2$ is induction on n, which will lead to the following two proof goals

Base Case: prove $\exists r\; r^2{\le}0 \wedge 0{<}(r{+}1)^2$
Induction Step: prove $\exists r\; r^2{\le}n{+}1 \wedge n{+}1{<}(r{+}1)^2$ assuming $\exists r_n\; r^2{\le}n \wedge n{<}(r_n{+}1)^2$.

The base case can be solved by choosing $r = 0$ and using standard arithmetical reasoning to prove the resulting proof obligation $0^2{\le}0 \wedge 0{<}(0{+}1)^2$.

In the induction step, one has to analyze the root r_n. If $(r_n{+}1)^2{\le}n{+}1$, then choosing $r = r_n{+}1$ will solve the goal. Again, the proof obligation $(r_n{+}1)^2{\le}n{+}1 \wedge n{+}1{<}((r_n{+}1){+}1)^2$ can be shown by standard arithmetical reasoning. $(r_n{+}1)^2 > n{+}1$, then one has to choose $r = r_n$ and prove $r_n^2{\le}n{+}1 \wedge n{+}1{<}(r_n{+}1)^2$ using standard arithmetical reasoning.

Figure A.1 shows the trace of a formal proof in the Nuprl system [40, 10] that uses exactly this line of argument. It initiates the induction by applying the library theorem

```
NatInd          ∀P:ℕ→ℙ.  (P(0) ∧ (∀i:ℕ⁺.  P(i-1) ⇒ P(i)))    ⇒   (∀i:ℕ.  P(i))
```

The base case is solved by assigning 0 to the existentially quantified variable and using Nuprl's autotactic (trivial standard reasoning) to deal with the remaining proof obligation. In the step case (from $i{-}1$ to i) it analyzes the root r for $i{-}1$, introduces a case distinction on $(r{+}1)^2{\le}i$ and then assigns either r or $r{+}1$, again using Nuprl's autotactic on the rest of the proof.

Nuprl is capable of extracting an algorithm from the formal proof, which then may be run within Nuprl's computation environment or be exported to other programming systems. The algorithm is represented in Nuprl's extended lambda calculus.

Depending on the formalization of the existential quantifier there are two kinds of algorithms that may be extracted. In the standard formalization, where $\exists$ is represented as a (dependent) product type, the algorithm – shown on the left* – computes both the integer square root r of a given natural number n and a proof term, which verifies that r is in fact the integer square root of n. If $\exists$ is represented as a set type, this verification information is dropped during extraction and the algorithm – shown on the right – only performs the computation of the integer square root.

*The place holders pf_k represent the actual proof terms that are irrelevant for the computation.

```
∀n:ℕ.  ∃r:ℕ.  r²≤n<(r+1)²
BY  allR
  n:ℕ
  ⊢ ∃r:ℕ.  r²≤n<(r+1)²
  BY  NatInd 1
  .....basecase.....
      ⊢ ∃r:ℕ.  r²≤0<(r+1)²
  √  BY  existsR ⌜0⌝ THEN Auto
  .....upcase.....
      i:ℕ⁺,  r:ℕ,  r²≤i-1<(r+1)²
      ⊢ ∃r:ℕ.  r²≤i<(r+1)²
      BY  Decide ⌜(r+1)²≤i⌝ THEN Auto
      .....Case 1.....
          i:ℕ⁺,  r:ℕ,  r²≤i-1<(r+1)²,  (r+1)²≤i
          ⊢ ∃r:ℕ.  r²≤i<(r+1)²
      √  BY  existsR ⌜r+1⌝ THEN Auto'
      .....Case 2.....
          i:ℕ⁺,  r:ℕ,  r²≤i-1<(r+1)²,  ¬((r+1)²≤i)
          ⊢ ∃r:ℕ.  r²≤i<(r+1)²
      √  BY  existsR ⌜r⌝ THEN Auto
```

Figure A.1: Proof of the Specification Theorem using Standard Induction.

```
let rec sqrt i
= if i=0 then <0,pf_0>
  else let <r,pf_{i-1}>  = sqrt (i-1)
       in
         if (r+1)²≤n  then <r+1,pf_i>
         else <r,pf_i'>
```

```
let rec sqrt i
= if i=0 then 0
  else let r = sqrt (i-1)
       in
         if (r+1)²≤n  then r+1
         else r
```

Using standard conversion mechanisms, Nuprl can then transform the algorithm into any programming language that supports recursive definition and export it to the corresponding programming environment. As this makes little sense for algorithms containing proof terms, we only convert the algorithm on the right. A conversion into SML, for instance, yields the following program.

```
fun sqrt n = if n=0 then 0
             else let val r = sqrt (n-1)
                  in
                    if n<(r+1)^2  then r
                    else r+1
                  end
```

A.2 Deriving an Algorithm that runs in $\mathcal{O}(\sqrt{n})$

Due to the use of standard induction on the input variable, the algorithm derived in the previous section is linear in the size of the input n, which is reduced by 1 in each step. Obviously, this is not the most efficient way to compute an integer square root. In the following we will derive more efficient algorithms by proving $\forall n\,\exists r\; r^2{\leq}n \wedge n{<}(r{+}1)^2$ in a different way. These proof, however, will have to rely on more complex induction schemes to ensure a more efficient computation.

A more common method to compute the integer square root of a given number n is to start a search for a possible result r. One starts with $r{=}0$ and then increases r until $(r{+}1)^2 > n$. In the context of a proof, this means that we need to introduce an auxiliary variable k for the search and perform induction on this variable instead of n.

```
    ∀n:ℕ.  ∃r:ℕ.  r²≤n<(r+1)²
    BY  allR  THEN  Assert ⌈∀j:ℕ.  (n-j)²≤n  ⇒  ∃r≥n-j. r²≤n<(r+1)²⌉
    .....Assertion.....
       n:ℕ,  j:ℕ,  (n-j)²≤n
       ⊢ ∃r≥n-j. r²≤n<(r+1)²
       BY  NatInd 2
       .....basecase.....
          n:ℕ,  (n-0)²≤n
          ⊢ ∃r≥n-0. r²≤n<(r+1)²
       √ BY  existsR ⌈n⌉ THEN Auto'
       .....upcase.....
          n:ℕ,  j:ℕ⁺,  (n-(j-1))²≤n  ⇒  ∃r≥n-(j-1). r²≤n<(r+1)²,  (n-j)²≤n
          ⊢ ∃r≥n-j. r²≤n<(r+1)²
          BY  Decide ⌈n<(n-j+1)²⌉ THEN Auto
          .....Case 1.....
             n:ℕ,  j:ℕ⁺,  (n-(j-1))²≤n  ⇒  ∃r≥n-(j-1). r²≤n<(r+1)²,  (n-j)²≤n,
             n<(n-j+1)²
             ⊢ ∃r≥n-j. r²≤n<(r+1)²
          √ BY  existsR ⌈n-j⌉ THEN Auto'
          .....Case 2.....
             n:ℕ,  j:ℕ⁺,  (n-(j-1))²≤n  ⇒  ∃r≥n-(j-1). r²≤n<(r+1)²,  (n-j)²≤n
             ¬(n<(n-j+1)²)
             ⊢ ∃r≥n-j. r²≤n<(r+1)²
             BY  impL 3 THEN Auto
               n:ℕ,  j:ℕ⁺,  (n-(j-1))²≤n  ⇒  ∃r≥n-(j-1). r²≤n<(r+1)²,  (n-
j)²≤n
               ¬(n<(n-j+1)²)
               ⊢ ∃r≥n-j. r²≤n<(r+1)²
           √ BY  existsR ⌈r⌉ THEN Auto'
    .....Main.....
       n:ℕ,  ∀j:ℕ.  (n-j)²≤n  ⇒  ∃r≥n-j. r²≤n<(r+1)²
       ⊢ ∃r:ℕ. r²≤n<(r+1)²
       BY  allL 2 ⌈n⌉ THEN Auto
          n:ℕ,  r:ℕ,  r≥n-n,  r²≤n<(r+1)²
          ⊢ ∃r:ℕ. r²≤n<(r+1)²
    √    BY  existsR ⌈r⌉ THEN Auto
```

Figure A.2: Proof of the Specification Theorem using Search

A naive approach would be to prove the theorem $\forall n\, \forall k\, \exists r{\geq}k\;\; r^2{\leq}n \wedge n{<}(r{+}1)^2$ using induction on k and then to instantiate this theorem with $k{=}0$. This approach, however, has two major flaws. First, the induction on k expresses a solution for k in terms of a solution for $k{-}1$, which is less efficient than a forward search. Second, the search must begin at some $k{>}\sqrt{n}$ but the theorem obviously does not hold for $k{>}\sqrt{n}$.

To fix these problems, we need to change the direction of the search into one that starts at 0 and recursively solves the problem for k by consulting a solution for $k{+}1$ until the square root has been found, which can be expressed by a standard induction over $j = n{-}k$. We also need to add a limit to the search, i.e. $(n{-}j)^2{=}k^2{\leq}n$.

The formal Nuprl proof begins by asserting $\forall j\ (n{-}j)^2{\leq}n \Rightarrow \exists r{\geq}(n{-}j)\ \ r^2{\leq}n \wedge n{<}(r{+}1)^2$, proves this statement by induction, and then instantiates it with $j{=}n$. Extracting the algorithm from this proof, depicted in Figure A.2, and converting it into SML leads to the following program, which now runs in $\mathcal{O}(\sqrt{n})$.

```
fun sqrt n = let fun aux j =
                   if j=0  then n
                   if n < (n-j+1)^2  then n-j
                   else aux (j-1)
             in
               aux n
             end
```

Note that the case $j{=}0$ is never reached unless n is 0.

By using different induction schemes it is possible to modify this algorithm into a more conventional form that uses an auxiliary variable k that is increasing instead of the term $n{-}j$, where j is decreasing. This induction scheme, however, needs to make explicit that choices for the auxiliary variable have an upper bound (i.e. n), whereas the lower bound zero is implicit in the other induction schemes that quantify over natural numbers. The induction scheme

```
RevNatInd     ∀P:ℕ→ℙ.  (∀i:{...t}.  (∀j:{i+1..t}.P(j)) ⇒ P(i))   ⇒   (∀i:{...t}.P(i))
```

which can easily be derived from the scheme `NatInd`, enables us to begin our proof by asserting $\forall k\ k^2{\leq}n \Rightarrow \exists r{\geq}k\ r^2{\leq}n \wedge$ and then to proceed as in Figure A.2, replacing every occurrence of $n{-}j$ by k. The extracted algorithm would now be

```
fun sqrt n = let fun aux k =
                   if k=n  then n
                   if n < (k+1)^2  then y
                   else aux (k+1)
             in
               aux 0
             end
```

Actually, the search algorithm is an instance of a generic search method that is implicitly contained in the following theorem

```
NatSearch     ∀P:ℕ→ℙ.  ∀n:ℕ.  P(n) ⇒ (∃k:{0..n}.  P(k) ∧ (∀j:{0..k-1}.  ¬P(j)))
```

which states the (bounded) existence of a minimal k with some property P. Instantiating this theorem with $P(k)$ replaced by $(k+1)^2 > n$ immediately gives us the desired search algorithm.

A.3 Deriving a Logarithmic Algorithm

One of the most efficient forms of computation on numbers is to operate on their binary representation and to construct a value bit by bit. The corresponding induction scheme requires proving a conclusion $P(x)$ from an induction hypothesis $P(x{\div}2)$, where $\div$ denotes integer division. For the integer square root problem, this induction scheme would have to be used on the *output variable* similarly to the way linear induction was used in the previous section.

It is much easierr, however, to use *4-adic induction* on the input variable instead, as this leads to a simpler proof. In fact, it is possible to mirror the proof given in Section A.1 by applying the library theorem

```
NatInd4            ∀P:ℕ→ℙ.  (P(0) ∧ (∀i:ℕ.  P(i÷4) ⇒ P(i)))    ⇒   (∀i:ℕ.  P(i))
```

and then replacing every occurrence of `r` in the arguments of a proof tactic by `2*r`. Apart from these differences, which are emphasized in both proofs, the proof in Figure A.3 is identical to the one in Figure A.1. Accordingly, the generated algorithms have exactly the same structure. Extracting the algorithm

```
∀n:ℕ.  ∃r:ℕ.  r² ≤ n < (r+1)²
BY  allR
  n:ℕ
  ⊢ ∃r:ℕ.  r² ≤ n < (r+1)²
  BY  NatInd4 1
  .....basecase.....
      ⊢ ∃r:ℕ.  r² ≤ 0 < (r+1)²
  √  BY  existsR ⌜0⌝ THEN Auto
  .....upcase.....
      i:ℕ,  r:ℕ,  r² ≤ i÷4 < (r+1)²
      ⊢ ∃r:ℕ.  r² ≤ i < (r+1)²
      BY  Decide ⌜((2*r)+1)² ≤ i⌝ THEN Auto
      .....Case 1.....
          i:ℕ,  r:ℕ,  r² ≤ i÷4 < (r+1)²,  ((2*r)+1)² ≤ i
          ⊢ ∃r:ℕ.  r² ≤ i < (r+1)²
      √  BY  existsR ⌜(2*r)+1⌝ THEN Auto'
      .....Case 2.....
          i:ℕ,  r:ℕ,  r² ≤ i÷4 < (r+1)²,  ¬(((2*r)+1)² ≤ i)
          ⊢ ∃r:ℕ.  r² ≤ i < (r+1)²
      √  BY  existsR ⌜2*r⌝ THEN Auto
```

Figure A.3: Proof of the Specification Theorem using Binary Induction

from the proof in Figure A.3 and converting it into SML leads to the following program, which now runs in *logarithmic time* (assuming that division by 4 is implemented as bit-shift operation).

```
fun sqrt n = if n=0 then 0
             else let val r = sqrt (n/4)
                  in
                    if n < (2*r+1)^2  then 2*r
                    else 2*r+1
                  end
```

Final Remarks

The algorithms and derivations presented in this note are contained in Nuprl's formal digital library that is now available online for interactive browsing at `http://www.nuprl.org`.

Using the proof strategies for inductive reasoning described in [74] it is possible to automatically construct all the proofs presented here. The implementation of this method as well as the proofs generated by it will be posted as part of the formal digital library in the future.

Appendix B

Formal Derivation of an Algorithm for the Stamps Problem

by Robert Constable and Christoph Kreitz

B.1 Introduction

We found the first stamps problem in the book *Elements of Discrete Mathematics,* by C. L. Liu from 1985. Here is how Liu casts the problem:

> Suppose we have stamps of two different denominations, 3 cents and 5 cents. We want to show that it is possible to make up exactly any postage of 8 cents or more using stamps of these two denominations. Clearly, the approach of showing case by case how to make up postage of 8 cents, 9 cents, 10 cents, and so on, using 3-cent and 5-cent stamps will not be a fruitful one, because there is an infinite number of cases to be examined. Let us consider an alternative approach. We want to show that if it is possible to make up exactly a postage of n cents using 3-cent and 5-cent stamps, then it is also possible to make up exactly a postage of $n + 1$ cents using 3-cent and 5-cent stamps.

Those who know a bit of number theory will recognize the Bezout identity that given two relatively prime numbers (also called *coprime* numbers) a, b, then for any integer z there are integers u, v such that $z = u \cdot a + v \cdot b$.

We now consider the restriction that u and v are positive and that for any $n \geq a + b$ there are natural numbers u, v such that $n = u \cdot a + v \cdot b$. For which relatively prime a, b is this equation solvable? We call these *stamps pairs*.

B.2 Deriving Algorithms for The Basic Stamps Problem

C. L. Liu provides a simple inductive solution for the original stamps problem, by showing how to make up a postage of $n{+}1$ cents using 3-cent and 5-cent stamps once we know how to make up a postage of n cents.

> We examine two cases. Suppose we make up a postage of n cents using at least one 5-cent stamp. Replacing a 5-cent stamp by two 3-cent stamps will yield a way to make up a postage of $n{+}1$ cents. On the other hand, suppose we make up a postage of n cents using 3-cent stamps only. Since $k{\geq}8$, there must be at least three 3-cent stamps. Replacing three 3-cent stamps by two 5-cent stamps will yield a way to make up a postage of $n{+}1$ cents.

```
⊢ ∀n:ℕ. n≥8 ⇒ ∃i,j:ℕ. n = i*3+j*5
BY  NatIndStartingAt ⌈8⌉
.....basecase.....
    ⊢ ∃i,j:ℕ. 8 = i*3+j*5
√  BY  ExR [⌈1⌉;⌈1⌉] THEN Auto
.....upcase.....
    n:ℕ, n>8, i:ℕ,  j:ℕ,  n−1 = i*3+j*5  ⊢  ∃i,j:ℕ. n = i*3+j*5
    BY  Decide ⌈j=0⌉ THEN Auto
    .....Case 1.....
        n:ℕ, n>8, i:ℕ,  j:ℕ,  n−1 = i*3+j*5, j=0  ⊢  ∃i,j:ℕ. n = i*3+j*5
    √  BY  ExR [⌈i−3⌉;⌈2⌉] THEN Auto'
    .....Case 2.....
        n:ℕ, n>8, i:ℕ,  j:ℕ,  n−1 = i*3+j*5, j≠0  ⊢  ∃i,j:ℕ. n = i*3+j*5
    √  BY  ExR [⌈i+2⌉;⌈j−1⌉]  THEN Auto'
```

Figure B.1: Inductive Proof of the Specification Theorem for the Basic Stamps Problem.

Figure B.1 shows the trace of a formal proof in the Nuprl system that uses exactly this line of argument. The stamps problem is formalized as the theorem

```
∀n:ℕ. n≥8 ⇒ ∃i,j:ℕ. n = i*3+j*5
```

and proven by induction starting at the value 8, for which we apply the library theorem

```
NatIndStartingAt   ∀k:ℕ. ∀P:ℕ→ℙ. (P(k) ∧ (∀i>k ⇒ P(i-1) ⇒ P(i))) ⇒ (∀i≥k. P(i))
```

The base case is solved by assigning 1 to both existentially quantified variables and using Nuprl's autotactic (trivial standard reasoning) to deal with the remaining proof obligation. In the step case from $n{-}1$ to n it analyzes the assignments i and j for $n{-}1$, introduces a case distinction on $j{=}0$ and then assigns either $i{-}3$ and 2 or $i{+}2$ and $j{-}1$, again using the autotactic to complete the proof.

The above proof implicitly contains an algorithm for computing the number of 3-cent and 5-cent stamps needed to make up a given postage n. Nuprl is capable of extracting this algorithm from the formal proof, and to execute it within Nuprl's computation environment or to export it to other programming systems.

Depending on the formalization of the existential quantifier there are two kinds of algorithms that may be extracted. If ∃ is represented as a (dependent) product type, the algorithm returns both the solution and a that verifies it. If ∃ is represented as a set type, this verification information is dropped during extraction and the algorithm – represented in Nuprl's extended lambda calculus and shown on the left – only performs the required computation. Using standard conversions, Nuprl can then transform the algorithm into any programming language that supports recursive definition and export it to the corresponding programming environment. A conversion into SML, for instance, yields the program shown on the right.

```
let rec stamps_assign n
= if n=8 then <1,1>
  else let <i, j> = stamps_assign (n-1)
       in
         if j=0  then <i-3, 2>
         else <i+2, j-1>
```

```
fun stamps_assign n
= if n=8 then 1,1
  else let val i,j = stamps_assign (n-1)
       in
         if j=0  then i-3, 2
         else i+2, j-1
       end
```

```
⊢ ∀n:ℕ. n≥8  ⇒  ∃i,j:ℕ. n = i*3 + j*5
BY  NatIndThreeStepStartingAt ⌜8⌝
.....basecase 1.....
    ⊢ ∃i,j:ℕ. 8 = i*3 + j*5
√  BY  ExR [⌜1⌝;⌜1⌝] THEN Auto
.....basecase 2.....
    ⊢ ∃i,j:ℕ. 9 = i*3 + j*5
√  BY  ExR [⌜3⌝;⌜0⌝] THEN Auto
.....basecase 3.....
    ⊢ ∃i,j:ℕ. 10 = i*3 + j*5
√  BY  ExR [⌜0⌝;⌜2⌝] THEN Auto
.....upcase.....
    n:ℕ, n≥8+3, i:ℕ,  j:ℕ,  n−3 = i*3 + j*5  ⊢  ∃i,j:ℕ. n = i*3 + j*5
√  BY  ExR [⌜i+1⌝;⌜j⌝] THEN Auto
```

Figure B.2: Solution of the Basic Stamps using 3-Step Induction.

Using stepwise induction is not the only way to solve the stamps problem. Instead of providing a solution for $n{=}8$ and then showing how to make up a postage of $n{+}1$ cents once we know how to do so for n cents, we could provide a solution for $n = 8$, 9, and 10, and then make up a postage of $n{+}3$ cents by adding a 3-cent stamp to the solution for n cents. In the formal proof, shown in Figure B.2, we have to use 3-Step induction for this purpose, again applying a library theorem. The resulting algorithm, shown in SML notation below, has the advantage that the loop computes much faster, as it does not involve a test and reduces n by 3 instead of 1.

```
fun stamps_assign n
= if n=8  then 1,1
  if n=9  then 3,0
  if n=10 then 0,2
  else let val i,j = stamps_assign (n-3)
       in
          i+1, j
       end
```

Since Nuprl's type theory comes with built-in division and quotient remainder functions, we can provide an even faster, non-inductive solution for the stamps problem. As before, we reduce a solution for n to the cases 8, 9, and 10, but we don't reduce n recursively, but do it in one step by computing $r = 8 + (n{-}8)$ rem 3. Given a solution i and j for r, the solution for n is then $i + (n{-}8){\div}3$ and j. A formal proof of this argument is given in Figure B.3. We assert that the problem has a solution over the limited range $8{\leq}n < 11$, provide a solution for each of these cases, and reduce the general problem by instantiating it with $r = 8{+}(n{-}8)$ rem 3 and then modify its solution by adding $(n-8){\div}3$ to i. As checking the solution involves reasoning about division and quotient remainder we supply a lemma to enable the autotactic to complete the proof. The resulting algorithm, shown in SML notation below, provides the fastest possible solution for the stamps problem.

```
fun stamps_assign n
= let q = (n-8)÷3
  and r = (n-8) rem 3 + 8
  in
     if r=8  then 1+q, 1
     if r=9  then 3+q, 0
```

```
⊢ ∀n:ℕ. n≥8 ⇒ ∃i,j:ℕ. n = i*3+j*5
BY Assert ⌈∀n:ℕ. 11>n≥8 ⇒ ∃i,j:ℕ. n=i*3+j*5⌉  THEN Auto
   .....Assertion.....
   n:ℕ, 11>n≥8  ⊢  ∃i,j:ℕ. n = i*3+j*5
   BY Choices [⌈n=8⌉; ⌈n=9⌉;⌈n=10⌉]
      .....Case n=8.....
         ⊢ ∃i,j:ℕ. 8 = i*3+j*5
     √  BY  ExR [⌈1⌉;⌈1⌉] THEN Auto
      .....Case n=9.....
         ⊢ ∃i,j:ℕ. 9 = i*3+j*5
     √  BY  ExR [⌈3⌉;⌈0⌉] THEN Auto
      .....Case n=10....
         ⊢ ∃i,j:ℕ. 10 = i*3+j*5
     √  BY  ExR [⌈0⌉;⌈2⌉] THEN Auto

   .....Reduction.....
   n:ℕ, n≥8, ∀n:ℕ.11>n≥8 ⇒ ∃i,j:ℕ.n=i*3+j*5  ⊢  ∃i,j:ℕ. n = i*3+j*5
   BY allL (-1) ⌈8+(n-8) rem 3⌉ THEN Repeat (exL (-1))
      n:ℕ, n≥8, i:ℕ, j:ℕ, 8+(n-8) rem 3 = i*3+j*5  ⊢  ∃i,j:ℕ. n =
i*3+j*5
   √  BY  ExR [⌈i+(n-8)÷3⌉;⌈j⌉] THEN ILemma `div_rem_sum` [⌈n-8⌉; ⌈3⌉
```

Figure B.3: Solution of the Basic Stamps using Direct Reduction.

```
if r=10 then 0+q, 2
```

B.3 An Informal Proof for the General Stamps Problem

In the previous section we have shown how to solve the stamps problem efficiently for the pair 3 and 5. Now the question is if there are other combinations of a and b that can be proven to be stamps pairs. Obviously, $a = 1$ and any b will be stamps pairs and so will be $a = 2$ and any odd number b. But are there others?

An informal solution for this problem was first presented at the International Summer School at Marktoberdorf in July 1995. Using basic number theory it shows that there cannot be any other stamps pairs. The statement and its proof are the following.

Let $a, b \in \mathbb{N}$ and without loss of generality $a < b$. If for all $n \geq a + b$ there are $i, j \in \mathbb{N}$ such that $n = i{\cdot}a + j{\cdot}b$ then a=1 or a=2 and b is odd or a=3 and b=5.

Proof: If $a = 1$, we're done, so assume $1 < a < b$

Since $a{+}b{+}1 = i{\cdot}a{+}j{\cdot}b$ for some i, j it must be that $a \mid (b{+}1)$ or $b{=}a{+}1$ (1)

Since $a{+}b{+}2 = i{\cdot}a{+}j{\cdot}b$ for some i, j it must be that $a{=}2$ or $a \mid (b{+}2)$ or $b{=}a{+}2$ (2)

Case analysis

$a = 2$**:** by (1), b must be odd

$a > 2$**:** then $b > 3$. We use (1) to split into subcases

$a \mid (b + 1)$**:** Then, because of $a > 2$, a cannot divide $b{+}2$ as well.
By (2), we thus have $b = a{+}2$.
Now, since $a{+}b{+}3 = i{\cdot}a{+}j{\cdot}b$ for some i, j we know $a{=}3$ or $a \mid (b{+}3)$ or $b{=}a{+}3$.
$b = a{+}3$ is impossible since $b = a{+}2$.
$a \mid (b{+}3)$ is impossible since $a \mid (b + 1)$ and $a > 2$.

Thus $a = 3$ and $b = 5$.

$b \,|\, (a+1)$**:** then by the same argument $b = a + 1$

But then by (2), $a \,|\, (a+3)$ or $a+1 \,|\, (a+2)$, both of which are impossible.

B.4 A Formal Proof for the General Stamps Problem

Although the above solution for the stamps problem was generally accepted, an attempt to recast this proof in a formal setting failed, since the argument for the case $b \,|\, (a+1)$ did not provide sufficient detail to complete the formal proof. In fact, being forced to take a closer look at this case revealed that the argument was wrong: the subcase $a \,|\, (a+3)$ is not impossible, but leads to another stamps pair, namely a=3 and b=4. But the formal proof also showed that there were no further stamps pairs.

Figure B.4 describes the main part of the formal proof. The proof proceeds by decomposing the proof goal using Nuprl's autotactic. In the case where we want to prove that there are only four combinations of stamps for which the stamps problem can be solved we consider three alternatives, among which the first (a=1) trivially leads to a solution and the other two are solved by instantiating separate lemmas with the tactic `ILemma`. In the other case, where we have to prove that the 4 combinations actually lead to a solution of the stamps problem, we do case analysis over the four possibilities, perform backward reasoning over a lemma to reduce the problem to the base case of the induction, and then provide explicit solutions for all possible values in the range $\{a{+}b...2{\cdot}a{+}b^{-}\}$. In the case where b is odd, we make use of the fact that an odd number is equal to $2{\cdot}c{+}1$ for some c.

The proofs of the main theorem and the lemmas use notation that extends the basic type theory of Nuprl to make the formal statements more comprehensible. For this purpose, the following *abstractions* were added to the library of the Nuprl system.

```
ABS int_upper     {i..}     ≡ {j:ℤ| i≤j}
ABS int_seg       {i..j⁻}   ≡ {k:ℤ| i≤k<j}
ABS divides        a | b     ≡ ∃c:ℤ. a = b*c
ABS is_odd        a is odd  ≡ 2 | a+1
ABS stampspairs    a and b are stamps pairs  ≡  ∀n:{a+b...}. ∃i,j:ℕ. n = i*a+j*b
```

Figure B.5 describes the proof of the lemma `stampspairs_properties`, which is used to reduce the stamps property to a problem over the finite range $\{a{+}b...2{\cdot}a{+}b^{-}\}$. Usually, one would prove this lemma by induction over the value `n`. However, since division ($i{\div}j$) and quotient remainder (i rem j) are primitives of Nuprl's type theory, we can provide a direct solution to the general problem by instantiating the limited one with an appropriate value. This requires us to show that $(((n{-}(a{+}b)){\div}a{+}i){\cdot}a){+}j{\cdot}b$ is in fact the same as the value n. As reasoning about division and quotient remainder is more complex than the autotactic can handle, we have to supply a lemma to make it complete the proof.

Figure B.6 shows the proof of lemma `stampspairs_if_two`, which is used to solve one of the cases of the main theorem. It states that a number b must be odd if 2 and b are stamps pairs. We prove it by instantiating the stamps property for the value $2{\cdot}b{+}1$ and then use arithmetical reasoning with the help of a lemma about division.

The most demanding proof in our solution is the one of lemma `stampspairs_if_greater_two`, shown in Figures B.7 and B.8. It shows that if $a > 2$ and $b > a$ are stamps pairs, then a must be 3 and b must be either 4 or 5. Essentially we follow the informal argument and state that a divides $b{+}1$ or $b{=}a{+}1$ and that a divides $b{+}2$ or $b{=}a{+}2$.

We prove the first claim by instantiating the stamps property for the value $a{+}b{+}1$ and then analyze how often b may have been used to create this sum. If b is not used, a must divide $b{+}1$. If b is used twice, $b{=}a{+}1$ must be the case. All other cases are impossible. For the second claim, we use a similar argument, this time with the value $a{+}b{+}2$.

```
    THM Stamps Theorem

    ∀a,b:ℕ.  (0<a ∧ a<b) ⇒
    a and b are stamps pairs  ⇔  a=1 ∨ (a=2 ∧ b is odd) ∨ (a=3 ∧ b=4) ∨
(a=3 ∧ b=5))
    BY Auto

       .....⇒.....
       a:ℕ, b:ℕ, 0<a, a<b, a and b are stamps pairs
       ⊢ a=1 ∨ (a=2 ∧ b is odd) ∨ (a=3 ∧ b=4) ∨ (a=3 ∧ b=5)
       BY Alternatives [⌜a=1⌝; ⌜a=2⌝; ⌜a>2⌝]

          .....Case 2.....
          a:ℕ, b:ℕ, 0<a, a<b, 2 and b are stamps pairs, a=2
          ⊢ a=1 ∨ (a=2 ∧ b is odd) ∨ (a=3 ∧ b=4) ∨ (a=3 ∧ b=5)
       √ BY ILemma ‘stampspairs_if_two‘ [⌜b⌝] THEN prover

          .....Case 3.....
          a:ℕ, b:ℕ, 0<a, a<b, a and b are stamps pairs, a>2
          ⊢ a=1 ∨ (a=2 ∧ b is odd) ∨ (a=3 ∧ b=4) ∨ (a=3 ∧ b=5)
       √ BY ILemma ‘stampspairs_if_greater_two‘ [⌜a⌝;⌜b⌝]

       .....⇐.....
       a:ℕ, b:ℕ, 0<a, a<b,  a=1 ∨ (a=2 ∧ b is odd) ∨ (a=3 ∧ b=4) ∨ (a=3 ∧
b=5)
       ⊢ a and b are stamps pairs
       BY AnalyzeCasesInHypothesis 5 THEN BackLemma ‘stampspairs_properties‘

          .....Case 1.....
          b:ℕ, 1<b,     n:{1+b..2*1+b⁻}  ⊢  ∃i,j:ℕ.  n = i*1 + j*b
       √ BY ExR [⌜n⌝; ⌜0⌝]

          .....Case 2.....
          b:ℕ, b is odd, n:{2+b..2*2+b⁻}  ⊢  ∃i,j:ℕ.  n = i*2 + j*b
       √ BY Choices [⌜n=2+b⌝    ; ⌜n=3+b⌝                    ]
             THENL   [ExR [⌜1⌝;⌜1⌝]; DVars [‘c’] 2 THEN ExR [⌜1 + c⌝;⌜0⌝]]

          .....Case 3.....
          n:{3+4..2*3+4⁻}                  ⊢  ∃i,j:ℕ.  n = i*3 + j*4
       √ BY Choices [⌜n=7⌝       ; ⌜n=8⌝      ;⌜n=9⌝        ]
             THENL   [ExR [⌜1⌝;⌜1⌝]; ExR [⌜0⌝;⌜2⌝]; ExR [⌜3⌝;⌜0⌝] ]

          .....Case 4.....
          n:{3+5..2*3+5⁻}                  ⊢  ∃i,j:ℕ.  n = i*3 + j*5
       √ BY Choices [⌜n=8⌝       ; ⌜n=9⌝      ;⌜n=10⌝       ]
             THENL   [ExR [⌜1⌝;⌜1⌝]; ExR [⌜3⌝;⌜0⌝]; ExR [⌜0⌝;⌜2⌝] ]
```

Figure B.4: Proof of the Main Theorem.

```
  THM stampspairs_properties
  ∀a,b:ℕ. 0<a ⇒ (∀n:{a+b..2*a+b⁻}. ∃i,j:ℕ. n = i*a+j*b)  ⇒  a and b are
stamps pairs
  BY Unfold `stampspairs` 0 THEN Auto

     a:ℕ, b:ℕ, 0<a,  ∀n:{a+b..2*a+b⁻}.  ∃i,j:ℕ.  n = i*a+j*b,  n:{a+b...}
     ⊢ ∃i,j:ℕ.  n = i*a+j*b
     BY allL 4 ⌜a + b + (n-(a+b) rem a)⌝ THEN Repeat (exL (-1))

        a:ℕ, b:ℕ, 0<a, n:{a+b...}, i:ℕ, j:ℕ,  a + b + (n-(a+b) rem a) =
i*a+j*b
        ⊢ ∃i,j:ℕ.  n = i*a+j*b
        BY ExR [⌜(n-(a+b))÷a + i⌝; ⌜j⌝]

           a:ℕ, b:ℕ, 0<a, n:{a+b...}, i:ℕ, j:ℕ,  a + b + (n-(a+b) rem a) =
i*a+j*b
           ⊢ n = (((n-(a+b))÷a + i) * a) + j*b
        √ BY ILemma `div_rem_sum` [⌜n-(a+b)⌝; ⌜a⌝]
```

Figure B.5: Proofs of the Reduction Theorem (stampspairs_properties)

Using these two assertions gives us 4 cases, among which the case $b{=}a{+}1 \wedge b{=}a{+}2$ is impossible. In the other three cases we use the laws of divisibility to prove that $a \mid b{+}1 \wedge a \mid b{+}2$ gives us $a{=}1$ (a contradiction), $a \mid b{+}1 \wedge b{=}a{+}2$ gives us $a{=}3$ and $b{=}5$, and $b{=}a{+}1 \wedge a \mid b{+}2$ gives us $a{=}3$ and $b{=}4$.

The above proofs rely on a lemmas about multiplication, division, and orders, which can be found in Nuprl's standard library. The following lemmas were used.

```
    THM mul_bounds_1a                      ∀a,b:ℕ.           0 ≤ a*b
    THM mul_bounds_1b                      ∀a,b:ℕ⁺.          0 < a*b
    THM mul_preserves_lt                   ∀a,b:ℤ.∀n:ℕ⁺.    a < b ⇒ n*a < n*b
    THM mul_preserves_le                   ∀a,b:ℤ.∀n:ℕ.      a ≤ b ⇒ n*a ≤ n*b
    THM multiply_functionality_wrt_le      ∀i₁,i₂,j₁,j₂:ℕ.   i₁≤j₁ ⇒ i₂≤j₂ ⇒ i₁*i₂ ≤ j₁*j₂
    THM div_rem_sum                        ∀a:ℤ.∀n:ℤ⁻⁰.      a = (a ÷ n)*n + a rem n
    THM divisor_of_sub                     ∀a,b₁,b₂:ℤ.        a | b₁ ⇒ a | b₂ ⇒ a | (b₁-
b₂)
    THM divisor_bound                       ∀a:ℕ.∀n:ℕ⁺.      a | b  ⇒  a≤b
    THM odd_mul_cancel                     ∀a,b:ℤ.            a*b is odd ⇒ b is odd
```

```
     THM stampspairs_if_two
      ∀b:ℕ.  2 and b are stamps pairs  ⇒  b is odd
      BY Auto THEN StampsInstance 2 ⌜2*b+1⌝

         b:ℕ, i:ℕ, j:ℕ,  2*b+1 = i*2+j*b  ⊢  b is odd
         BY ILemma `odd_mul_cancel` [⌜j⌝;⌜b⌝]

            b:ℕ, i:ℕ, j:ℕ,  2*b+1 = i*2+j*b  ⊢  j*b is odd
         √ BY RepUnfolds ``is_odd divides`` 0 THEN ExR [⌜b-i + 1⌝]
```

Figure B.6: Proofs of the requirements for "good" stamps.

```
    THM stampspairs_if_greater_two
    ∀a,b:ℕ.  (2<a ∧ a<b) ⇒
    a and b are stamps pairs  ⇒  (a=3 ∧ b=4) ∨ (a=3 ∧ b=5))
    BY Auto THEN AssertCases ⌜(a|b+1 ∨ b=a+1) ∧ (a|b+2 ∨ b=a+2)⌝

        .....Assertion 1.....
        a:ℕ, b:ℕ, 2<a, a<b, a and b are stamps pairs ⊢  a|b+1 ∨ b=a+1
        BY StampsInstance 5 ⌜a+b+1⌝ THEN EqChoices [ ⌜j=0⌝; ⌜j=1⌝; ⌜j=2⌝; ⌜j>2⌝]

           .....Case j=0.....
           a:ℕ, b:ℕ, 2<a, a<b, i:ℕ, j:ℕ, a+b+1 = i*a+0*b  ⊢  a|b+1 ∨ b=a+1
        √ BY orR1 THEN DividesWitness ⌜i - 1⌝
           .....Case j=1.....
           a:ℕ, b:ℕ, 2<a, a<b, i:ℕ, j:ℕ, a+b+1 = i*a+1*b  ⊢  a|b+1 ∨ b=a+1
        √ BY Assert ⌜a|1⌝ THENL [DividesWitness ⌜i-1⌝; ILemma ‘divisor_bound‘
[⌜a⌝;⌜1⌝]]
           .....Case j=2.....
           a:ℕ, b:ℕ, 2<a, a<b, i:ℕ, j:ℕ, a+b+1 = i*a+2*b  ⊢  a|b+1 ∨ b=a+1
        √ BY EqChoices [⌜i=0⌝;⌜i>0⌝]  THENL [Auto’; ILemma ‘mul_bounds_1b‘
[⌜i⌝;⌜a⌝] ]
              % -----------------------------------------------------------
----- +
              | The second case uses the inequality ⌜0<i*a⌝ to show a contra-
diction |
              + -----------------------------------------------------------
----- %
           .....Case j>2.....
           a:ℕ, b:ℕ, 2<a, a<b, i:ℕ, j:ℕ, a+b+1 = i*a+j*b, j>2  ⊢  a|b+1 ∨
b=a+1
        √ BY % ----------------------------------------------------------
+
              | Show by a chain of inequalities that there is a contradiction
|
              + -----------------------------------------------------------
%
                  ILemma ‘mul_bounds_1a‘    [⌜i⌝;⌜a⌝]      % 0 ≤ i*a   %
              THEN ILemma ‘mul_preserves_lt‘ [⌜2⌝;⌜j⌝;⌜b⌝]  % b*2 < b*j %
```

Figure B.7: Proofs of the requirements for “good” stamps.

The proofs also employ a variety of reasoning *tactics* that were written to make the formal proof comprehensible. Tactics are metalevel programs that control the application of reasoning rules of a fundamental proof calculus. The tactics used in our proofs were written to mimic specific reasoning steps that a human would use in an argument by expressing them in terms of elementary proof rules. Because we chose memnonic names (and used comments in a the proofs), most of them should be self-explanatory.

```
          .....Assertion 2.....
          a:ℕ, b:ℕ, 2<a, a<b, a and b are stamps pairs  ⊢  a|b+2 ∨ b=a+2
       √ BY % -------------------------------------------------------------
---- +
            | This is almost identical to Assertion 1, so we do everything at
once |
            + --------------------------------------------------------------
---- %
                  StampsInstance 5 ⌜a+b+2⌝  THEN  EqChoices [ ⌜j=0⌝; ⌜j=1⌝;
⌜j=2⌝; ⌜j>2⌝]
            THENL [ orR1 THEN DividesWitness ⌜i - 1⌝
                  ; Assert ⌜a|2⌝ THENL [ DividesWitness ⌜i-1⌝
                                      ; ILemma `divisor_bound` [⌜a⌝;⌜2⌝]]
                  ; EqChoices [⌜i=0⌝;⌜i>0⌝]
                    THENL [ Auto'; ILemma `multiply_functionality_wrt_le`
[⌜1⌝;⌜2⌝;⌜i⌝;⌜a⌝]]
                         % ---------------------------------------------------
-- +
                         | The second case uses 1*2<i*a to show a contradic-
tion |
                         + ---------------------------------------------------
-- %
                  ;      ILemma `mul_bounds_1a`    [⌜i⌝;⌜a⌝]       % 0 ≤ i*a   %
                    THEN ILemma `mul_preserves_le` [⌜3⌝;⌜j⌝;⌜b⌝]   % b*3 ≤ b*j %
                  ]

          .....Asserted Case 1.....
          a:ℕ, b:ℕ, 2<a, a<b, a|b+1, a|b+2  ⊢  (a=3 ∧ b=4) ∨ (a=3 ∧ b=5)
       √ BY % -------------------------------------------------------------
----- +
            | Analyzing Hyps 5 and 6 gives us a=1, which contradicts hypothe-
sis 6   |
            + --------------------------------------------------------------
----- %
                 FwdLemma `divisor_of_sub` [6;5]
            THEN Subst ⌜(b+2 - (b+1))=1⌝ (-1) THENA Auto
            THEN ILemma `divisor_bound` [⌜a⌝;⌜1⌝]

          .....Asserted Case 2.....
          a:ℕ, b:ℕ, 2<a, a<b, a|b+1, b=a+2  ⊢  (a=3 ∧ b=4) ∨ (a=3 ∧ b=5)
       √ BY % ----------------------------------------- +
            | Analyzing Hyps 5 and 6 gives us  a=3 ∧ b=5 |
            + ----------------------------------------- %
             Assert ⌜a|3⌝ THENL [ DVars [`c'] 5 THEN DividesWitness ⌜c - 1⌝
                                ; ILemma `divisor_bound` [⌜a⌝;⌜3⌝] ]

          .....Asserted Case 3.....
          a:ℕ, b:ℕ, 2<a, a<b, b=a+1, a|b+2  ⊢  (a=3 ∧ b=4) ∨ (a=3 ∧ b=5)
       √ BY % ----------------------------------------- +
            | Analyzing Hyps 5 and 6 gives us  a=3 ∧ b=4 |
            + ----------------------------------------- %
             Assert ⌜a|3⌝ THENL [ DVars [`c'] 6 THEN DividesWitness ⌜c - 1⌝
                                ; ILemma `divisor_bound` [⌜a⌝;⌜3⌝] ]
```

Figure B.8: Proof of `stampspairs_if_greater_two` (continued)

Acknowledgements

I want to thank Stuart Allen and Mark Bickford for their contributions to these notes and Juanita Heyerman for preparing them in Latex under the pressure of deadlines.

Bibliography

[1] Uri Abraham. On interprocess communication and the implementation of multi-writer atomic registers. *Theoretical Computer Science*, 149:257–298, 1995.

[2] Uri Abraham. *Models for Concurrency*, volume 11 of *Algebra, Logic and Applications Series*. Gordon and Breach, 1999.

[3] Uri Abraham, Shlomi Dolev, Ted Herman, and Irit Koll. Self-stabilizing ℓ-exclusion. *Theoretical Computer Science*, 266:653–692, 2001.

[4] S. Abramsky. Proofs as processes. *Theoretical Computer Science*, 135(1):5–9, 1994.

[5] S. Abramsky. Process realizability. In F. L. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation: Proceedings of the 1999 Marktoberdorf Summer School*, pages 167–180. IOS Press, 2000.

[6] Peter Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 739–782. North-Holland, 1977.

[7] Peter Aczel. The type theoretic interpretation of constructive set theory: Inductive definition. In *Logic, Methodology and Philosophy of Science VII*, pages 17–49. Elsevier Science Publishers, 1986.

[8] Peter Aczel and Michael Rathjen. Notes on constructive set theory. Technical Report 40, Mittag-Leffler, 2000/2001.

[9] Stuart Allen, Mark Bickford, Robert Constable, et al. FDL: A prototype formal digital library. PostScript document on website, May 2002. `http://www.nuprl.org/html/FDLProject/02cucs-fdl.html`.

[10] Stuart Allen, Robert Constable, Richard Eaton, Christoph Kreitz, and Lori Lorigo. The Nuprl open logical environment. In McAllester [85], pages 170–176.

[11] Stuart F. Allen. *A Non-Type-Theoretic Semantics for Type-Theoretic Language*. PhD thesis, Cornell University, 1987.

[12] Stuart F. Allen. Abstract identifiers and textual reference. Technical Report TR2002-1885, Cornell University, Ithaca, New York, 2002.

[13] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication sub-system for high availability. In *22nd Annual International Symposium on Fault-Tolerant Computing*, pages 76–84, July 1992.

[14] Myla Archer, Constance Heitmeyer, and Elvinia Riccobene. Proving invariants of I/O automata with TAME. *Automated Software Engineering*, 9(3):201–232, 2002.

[15] A. Asperti, L. Padovani, C. Sacerdoti Coen, and I. Schena. HELM and the semantic math-web. In Boulton and Jackson [32], pages 59–74.

[16] Roland C. Backhouse, Paul Chisholm, Grant Malcolm, and Erik Saaman. Do-it-yourself type theory. *Formal Aspects of Computing*, 1:19–84, 1989.

[17] Andrew Barber, Philippa Gardner, Masahito Hasegawa, and Gordon D. Plotkin. From action calculi to linear logic. In Mogens Nielsen and Wolfgang Thomas, editors, *Computer Science Logic, 11th International Workshop, Annual Conference of the EACSL, Aarhus, Denmark, August 23-29, 1997, Selected Papers*, volume 1414 of *Lecture Notes in Computer Science*, pages 78–97. Springer, 1998.

[18] David A. Basin. An environment for automated reasoning about partial functions. In E. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, volume 310 of *Lecture Notes in Computer Science*, pages 101–110. Springer-Verlag, NY, 1988.

[19] J. L. Bates and Robert L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):53–71, 1985.

[20] H. Benl, U. Berger, H. Schwichtenberg, et al. Proof theory at work: Program development in the Minlog system. In W. Bibel and P. G. Schmitt, editors, *Automated Deduction*. Kluwer, 1998.

[21] U. Berger and H. Schwichtenberg. Program development by proof transformation. In Ulrich Berger and H. Schwichtenberg, editors, *Proof and Computation. Proceedings of the NATO Advanced Study Institute, Marktoberdorf, Germany*, volume 139 of *Series F: Computer and Systems Sciences*, pages 299–340, Berlin, 1995. Springer.

[22] G. Berry and G. Boudol. The chemical abstract machine. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages*, pages 81–94, 1990.

[23] Mark Bickford and Robert L. Constable. A logic of events. Tech Report TR2003-1893, Cornell University, 2003.

[24] Mark Bickford and Jason J. Hickey. Predicate transformers for infinite-state automata in Nuprl type theory. In *Proceedings of 3rd Irish Workshop in Formal Methods*, 1999.

[25] Mark Bickford, Christoph Kreitz, Robbert van Renesse, and Xiaoming Liu. Proving hybrid protocols correct. In Boulton and Jackson [32], pages 105–120.

[26] Ken Birman, Robert Constable, Mark Hayden, Jason Hickey, Christoph Kreitz, Robbert van Renesse, Ohad Rodeh, and Werner Vogels. The Horus and Ensemble projects: Accomplishments and limitations. In *DARPA Information Survivability Conference and Exposition (DISCEX 2000)*, pages 149–161, Hilton Head, SC, 2000. IEEE Computer Society Press.

[27] Kenneth P. Birman and Thomas A. Joseph. Exploiting virtual synchrony in distributed systems. *Proc 11th Symposium on Operating Systems Principles (SOSP)*, pages 123–138, November 1987.

[28] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comp. Syst.*, 5(1):47–76, February 1987.

[29] Kenneth P. Birman and Robbert van Renesse, editors. *The Isis Book: Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1994.

[30] E. Bishop. *Foundations of Constructive Analysis*. McGraw Hill, NY, 1967.

[31] E. Bishop and D. Bridges. *Constructive Analysis*. Springer, New York, 1985.

[32] Richard Boulton and Paul Jackson, editors. *14^{th} International Conference on Theorem Proving in Higher Order Logics*, volume 2152 of *Lecture Notes in Computer Science*, Edinburgh, Scotland, September 2001. Springer-Verlag.

[33] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[34] K. Mani Chandy and Carl Kesselman. CC++: A declarative concurrent object oriented programming notation. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, chapter 11, pages 281–313. MIT Press, Boston, 1993.

[35] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[36] Edmund M. Clarke and E. Allen Emerson. Synthesis of synchronization skeletons from branching time temporal logic. In *Proc. Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science (LNCS)*, pages 52–71. Springer–Verlag, 1982.

[37] W. Rance Cleaveland, editor. *5^{th} International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*. Springer, 1999.

[38] Robert L. Constable. Constructive mathematics and automatic program writers. In *Proceedings of the IFIP Congress*, pages 229–233. North-Holland, 1971.

[39] Robert L. Constable. Types in logic, mathematics and programming. In S. R. Buss, editor, *Handbook of Proof Theory*, chapter X, pages 683–786. Elsevier Science B.V., 1998.

[40] 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.

[41] Robert L. Constable and Jason Hickey. Nuprl's class theory and its applications. In Friedrich L. Bauer and Ralf Steinbrueggen, editors, *Foundations of Secure Computation*, NATO ASI Series, Series F: Computer & System Sciences, pages 91–116. IOS Press, 2000.

[42] Robert L. Constable and Scott F. Smith. Computational foundations of basic recursive function theory. *Theoretical Computer Science*, 121:89–112, December 1993.

[43] Thierry Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[44] Thierry Coquand and Christine Paulin-Mohring. Inductively defined types, preliminary version. In *COLOG '88, International Conference on Computer Logic*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer, Berlin, 1990.

[45] Peter Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 280–306. Cambridge University Press, 1991.

[46] E. Allen Emerson and Edmund M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.

[47] Kai Engelhardt, Ron van der Meyden, and Yoram Moses. A refinement theory that supports reasoning about knowledge and time for synchronous agents. In Robert Nieuwenhuis and Andrei Voronkov, editors, *8th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 2250 of *LNAI*, pages 125–141. Springer-Verlag, December 2001.

[48] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning About Knowledge*. Massachusetts Institute of Technology, 1995.

[49] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. Knowledge-based programs. *Distributed Computing*, 10(4):199–225, 1997.

[50] Harvey Friedman. Classically and intuitionistically provably recursive functions. In D. S. Scott and G. H. Muller, editors, *Higher Set Theory*, volume 699 of *Lecture Notes in Mathematics*, pages 21–28. Springer-Verlag, 1978.

[51] Michael Gordon and Tom Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, Cambridge, 1993.

[52] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: a mechanized logic of computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, NY, 1979.

[53] Joseph Y. Halpern. A note on knowledge-based programs and specifications. *Distributed Computing*, 13(3):145–153, 2000.

[54] Joseph Y. Halpern and Ronald Fagin. Modeling knowledge and action in distributed systems. *Distributed Computing*, 3(4):159–177, 1989.

[55] Joseph Y. Halpern and Riccardo Pucella. On the relationship between Strand spaces and multi-agent systems. In *Proceedings of the Eighth ACM Conference on Computer and Communications Security (CCS-8)*, pages 106–115, 2001.

[56] Joseph Y. Halpern and Richard A. Shore. Reasoning about common knowledge with infinitely many agents. In *Proceedings of the 14th IEEE Symposium on Logic in Computer Science*, pages 384–393, 1999.

[57] Peter Hancock and Anton Setzer. Interactive programs in dependent type theory. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic, 14th International Workshop, CSL 2000*, volume 1862 of *LNCS*. Springer-Verlag.

[58] John Harrison. High-level verification using theorem proving and formalized mathematics (extended abstract). In McAllester [85], pages 1–6.

[59] Mark Hayden. *The Ensemble System*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, NY, January 1998.

[60] Mark Hayden and Robbert van Renesse. Optimizing layered communication protocols. In *Proceedings of the High Performance Distributed Computing*, Portland, Oregon, August 1997.

[61] Jason Hickey, Aleksey Nogin, Robert L. Constable, Brian E. Aydemir, Eli Barzilay, Yegor Bryukhov, Richard Eaton, Adam Granicz, Alexei Kopylov, Christoph Kreitz, Vladimir N. Krupski, Lori Lorigo, Stephan Schmitt, Carl Witty, and Xin Yu. MetaPRL — A modular logical environment. Accepted to the TPHOLs 2003 Conference, 2003.

[62] Jason J. Hickey. *The MetaPRL Logical Programming Environment*. PhD thesis, Cornell University, Ithaca, NY, January 2001.

[63] Jason J. Hickey, Nancy Lynch, and Robbert Van Renesse. Specifications and proofs for Ensemble layers. In Cleaveland [37], pages 119–133.

[64] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[65] Amanda Holland-Minkley, Regina Barzilay, and Robert L. Constable. Verbalization of high-level formal proofs. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, pages 277–284. AAAI, July 1999.

[66] Douglas J. Howe. Importing mathematics from HOL into Nuprl. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics*, volume 1125, of *Lecture Notes in Computer Science*, pages 267–282. Springer-Verlag, Berlin, 1996.

[67] Douglas J. Howe. Semantic foundations for embedding HOL in Nuprl. In Martin Wirsing and Maurice Nivat, editors, *Algebraic Methodology and Software Technology*, volume 1101 of *Lecture Notes in Computer Science*, pages 85–101. Springer-Verlag, Berlin, 1996.

[68] Paul B. Jackson. *The Nuprl Proof Development System, Version 4.2 Reference Manual and User's Guide*. The PRL Group at Cornell University, 1996.

[69] D. A. Karr, C. Rodrigues, J.P. Loyall, R. E. Schantz, Y. Krishnamurthy, I. Pyarali, and D. C. Schmidt. Application of the QuO quality-of-service framework to a distributed video application. In *International Symposium on Distributed Objects and Applications*, 2001.

[70] Michael Kohlhase. OMDOC: An open markup format for mathematical docuemnts. Seki Report SR-00-02, Fachbereich Informatick, Universitat des Saarlandes, 2000. `http://www.mathweb.org/omdoc`.

[71] Alexei Kopylov. Dependent intersection: A new way of defining records in type theory. In *Proceedings of 18th IEEE Symposium on Logic in Computer Science*, pages 86–95, 2003.

[72] Christoph Kreitz. Automated fast-track reconfiguration of group communication systems. In Cleaveland [37], pages 104–118.

[73] Christoph Kreitz, Mark Hayden, and Jason J. Hickey. A proof environment for the development of group communications systems. In *Fifteen International Conference on Automated Deduction*, number 1421 in Lecture Notes in Artificial Intelligence, pages 317–332. Springer, 1998.

[74] Christoph Kreitz and Brigitte Pientka. Connection-driven inductive theorem proving. *Studia Logica*, 69(2):293–326, 2001.

[75] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Comms. ACM*, 21(7):558–65, 1978.

[76] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, Boston, 2003.

[77] Elizabeth I. Leonard and Constance L. Heitmeyer. Program synthesis from formal requirements specifications using apts. *Higher-Order and Symbolic Computation*, 2003. To appear.

[78] Xiaoming Liu, Christoph Kreitz, Robbert van Renesse, Jason J. Hickey, Mark Hayden, Kenneth Birman, and Robert Constable. Building reliable, high-performance communication systems from components. In *17th ACM Symposium on Operating Systems Principles (SOSP'99)*, volume 33(5) of *Operating Systems Review*, pages 80–92. ACM Press, December 1999.

[79] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[80] Nancy Lynch and Mark Tuttle. An introduction to Input/Output automata. *Centrum voor Wiskunde en Informatica*, 2(3):219–246, September 1989.

[81] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, Berlin, 1992.

[82] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, Berlin, 1995.

[83] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Trans. Program. Lang. and Syst.*, 6(1):68–93, 1984.

[84] Per Martin-Löf. Constructive mathematics and computer programming. In *Proceedings of the Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.

[85] David McAllester, editor. *Proceedings of the 17th International Conference on Automated Deduction*, volume 1831 of *Lecture Notes in Artificial Intelligence*. Springer Verlag, 2000.

[86] J. Meseguer and M.-O. Stehr. The HOL-Nuprl connection from the viewpoint of general logic. Working paper, June 1999.

[87] R. Milner. Action structures and the π-calculus. In Helmut Schwichtenberg, editor, *Proof and Computation*, volume 139 of *NATO Advanced Study Institute, International Summer School held in Marktoberdorf, Germany, July 20–August 1, 1993, NATO Series F*, pages 219–280. Springer, Berlin, 1994.

[88] Robin Milner. *Communication and Concurrency*. Prentice-Hall, London, 1989.

[89] Robin Milner. Calculi for interaction. *Acta Informatica*, 33(8):707–737, 1996.

[90] Evan Moran. *Adding Intersection and Union Types to Howe's Model of Type Theory [working title]*. PhD thesis, Cornell University, 2003.

[91] Chetan Murthy. An evaluation semantics for classical proofs. In *Proceedings of Sixth Symposium on Logic in Comp. Sci.*, pages 96–109. IEEE, Amsterdam, The Netherlands, 1991.

[92] Pavel Naumov. Importing Isabelle formal mathematics into Nuprl. In *Supplemental Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics, Nice, France*, September 1999.

[93] Tobias Nipkow. Structured Proofs in Isar/HOL. In H. Geuvers and F. Widijk, editors, *Types for Proofs and Programs (TYPES 2002)*, volume 2646 of *LNCS*, pages 259–278. Springer, 2003.

[94] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[95] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory*. Oxford Sciences Publication, Oxford, 1990.

[96] Kristen Nygaard and Ole-Johan Dahl. The development of the SIMULA languages. In Richard L. Wexelblat, editor, *History of Programming Languages*. Academic Press, 1981.

[97] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[98] Sam Owre and Natarajan Shankar. The formal semantics of PVS. Technical Report SRI-CSL-97-2, Computer Science Laboratory, SRI International, Menlo Park, CA, aug 1997.

[99] Lawrence C. Paulson. Isabelle*: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1994.

[100] Lawrence C. Paulson. Mechanized proofs for a recursive authentication protocol. In *10th Computer Security Foundations Workshop*, pages 84–95. IEEE Computer Society Press, 1997.

[101] Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the calculus of constructions. In *Mathematical Foundations of Program Semantics, 5th International Conference*, volume 442 of *Lecture Notes in Computer Science*, pages 209–228. Springer-Verlag, 1989.

[102] Frank Pfenning and Carsten Schürmann. Twelf — a meta-logical framework for deductive systems. In Harald Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 202–206, Berlin, July 7–10 1999. Trento, Italy.

[103] K. Rustan, M. Leino, and Greg Nelson. An extended static checker for modula-3. In Kai Koskimies, editor, *Compiler Construction: Seventh International Conference, CC'98*, volume 1383 of *Lecture Notes in Computer Science*, pages 302–305. Springer, April 1998.

[104] Fred B. Schneider. *On Concurrent Programming*. Springer-Verlag, New York, 1997.

[105] Anton Setzer. *Proof theoretical strength of Martin-Löf Type Theory with W-type and one universe*. PhD thesis, Ludwig-Maximilians-Universität, München, September 1993.

[106] Anton Setzer. Translating set theoretical proofs into type theoretical programs. In G. Gottlob, A. Leitsch, and D. Mundici, editors, *Computational Logic and Proof Theory*, volume 1289 of *LNAI*, pages 278–289. Springer-Verlag, 1997.

[107] Thomas Streicher. *Correctness and Completeness of a Categorical Semantics of the Calculus of Constructions*. PhD thesis, Universität Passau, 1988.

[108] F. J. Thayer, J. H. Herzog, and J. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 7(2/3):191–230, 1999.

[109] Laurent Théry. A machine-checked implementation of Buchberger's algorithm. *Journal of Automated Reasoning*, 26(2):107–137, February 2001.

[110] Anne Sjerp Troelstra. On the syntax of Martin-Löf's type theories. *Theoretical Computer Science*, 51:1–26, 1987.

[111] Robbert van Renesse, Kenneth P. Birman, Mark Hayden, Alexey Vaysburg, and David Karr. Building adaptive systems using Ensemble. *Software: Practice and Experience*, 28(9):963–979, July 1998.

[112] Robbert van Renesse, Takako Hickey, and Kenneth P. Birman. Design and performance of Horus: A lightweight group communications system. Department of Computer Science TR94-1442, Cornell University, Ithaca, NY, 1994.

[113] Walter P. van Stigt. *Brouwer's Intuitionism*. North-Holland, Amsterdam, 1990.

[114] M. Y. Vardi. An automata-theoretic approach to fair realizability and synthesis. In P. Wolper, editor, *Computer Aided Verification, Proceedings of the 7th International Conference*, volume 939 of *Lecture Notes in Computer Science*, pages 267–292. Springer-Verlag, 1995.

[115] G. Winskel. *Events in Computation*. PhD thesis, University of Edinburgh, 1980.

[116] G. Winskel. An introduction to event structures. In J. W. de Bakker et al., editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, number 345 in Lecture Notes in Computer Science, pages 364–397. Springer, 1989.

[117] N. Wirth and C.A.R. Hoare. A contribution to the development of ALGOL. *Communications of the ACM*, 9:413–432, 1966.

[118] N. Wirth and Helmut Weber. EULER: a generalization of ALGOL, and its formal definition: Part II. *Communications of the ACM*, 9:89–99, 1966.

[119] Job Zwiers, Willem P. de Roever, and Peter van Emde Boas. Compositionality and concurrent networks: Soundness and completeness of a proofsystem. In *ICALP 1985*, pages 509–519, 1985.