TPHOLs 2003 Tutorial
Introduction to MetaPRL Theorem Prover

Jason Hickey Nathaniel Gray Aleksey Nogin Cristian Ţăpuş
Xin Yu

September 29, 2003

Contents

1 Getting Started **2**
1.1 Installing MetaPRL on Linux . . . 2
1.1.1 Installing OCaml and OMake from RPMs . . . 2
1.1.2 Installing OCaml and OMake from source . . . 2
1.1.3 Copying MetaPRL . . . 3
1.2 Installing MetaPRL for Windows . . . 3
1.2.1 Installing Cygwin . . . 3
1.2.2 Customizing your environment . . . 4
1.2.3 Installing the remaining tools . . . 4
1.2.4 Building MetaPRL . . . 4
1.3 Configuring MetaPRL . . . 4

2 Introduction **7**

3 Defining a First-Order Logic **7**
3.1 Logical constants . . . 7
3.1.1 The `False` interface . . . 7
3.1.2 The `False` implementation . . . 8
3.1.3 The `True` constant . . . 9
3.2 Basic Automation . . . 10
3.2.1 Manual addition to the `dT` tactic . . . 12
3.2.2 Automatic addition of automation . . . 12
3.3 Adding the propositional connectives . . . 13
3.4 Defining negation . . . 19
3.5 Adding structural rules . . . 25
3.6 Defining a propositional logic . . . 29
3.7 Proof automation for constructive propositional logic . . . 33
3.8 The quantifiers . . . 36

4 Relating the propositional logic to type theory **41**
4.1 Defining well-formed formulas 41
4.2 Deriving FOL from type theory 44

5 Defining classical first-order logic **49**

6 Deriving classical first-order logic from the Nuprl type theory **50**

7 Conclusion **53**

Issues:

- Dyckoffs algorithm is implemented incorrectly
- Callig the quantifier domain "Pred" does not make sense
- Solutions use `prim_rw` for `unfold_not` intead of `define`
- `fol_theory.mli` is missing; should not include the fol_type in part1
- `distrib_or2` is proven by hand, not with propDecidT.
- The "part2" solutions should be identical to `theories/fol`

1 Getting Started

Before you install MetaPRL you need to install a few dependencies: a custom build of the OCaml compiler version 3.06 and the OMake build tool. Because MetaPRL requires a customized build of OCaml (it needs a number of files that are not built and/or installed by default), you will not be able to compile it with a standard OCaml installation.

1.1 Installing MetaPRL on Linux

For Linux installations you can either install MetaPRL from binary RPMs or from source.

1.1.1 Installing OCaml and OMake from RPMs

If you are using an RPM-Based Linux distribution then you can install OCaml and OMake from the binary RPMs included on the CD. Assuming your CD is mounted at /mnt/cdrom, and you are running RedHat 8.0 you must first use the `su` command to become root, then install the RPMs with:

```
rpm -Uvh /mnt/cdrom/RPMS/RHxxx/ocaml-3.0.6-xxx.rpm
rpm -Uvh /mnt/cdrom/RPMS/RHxxx/omake-xxx-xxx.rpm
```

Note: you need to use the OCaml RPM that was compiled for the same distribution that you are using. If you are using a Linux distribution for which we do not have an OCaml binary RPM, you should create your own by recompiling from the source RPM by running

```
rpmbuild --rebuild /mnt/cdrom/SPMS/RHxxx/ocaml-3.0.6-xxx.src.rpm
```

For OMake such exact distribution match is not as important.

1.1.2 Installing OCaml and OMake from source

If you cannot use the RPM packages we provide, you will need to build OCaml and OMake from source.

To build OCaml you first must copy it from the CD, then configure and make it:

```
cp -r /mnt/cdrom/tarballs/src/ocaml-3.0.6-xxx.tar.gz .
tar xzvf ocaml-3.0.6-xxx.tar.gz
cd ocaml
./configure
make
su
make install
exit
cd ..
```

Building OMake is very similar to building OCaml:

```
cp -r /mnt/cdrom/tarballs/src/omake-xxx-xxx.tar.gz .
tar xzvf omake-xxx-xxx.tar.gz cd omake
./configure
make world world.opt opt.opt
su
make install
exit
```

1.1.3 Copying MetaPRL

Next you need to copy the MetaPRL source directory from the CD to a local directory. For example, to copy it to your home directory you would type:

```
cp -r /mnt/cdrom/metaprl ~
```

You are now ready to configure MetaPRL. Please skip ahead to section 1.3.

1.2 Installing MetaPRL for Windows

For Windows, we will be using the Cygwin environment from Cygnus Solutions. MetaPRL *can* be built for native Win32, but the build requires Visual C++. Cygwin provides a Unix-like environment, including the X window system, and the build is very similar to that for Linux.

1.2.1 Installing Cygwin

You can install Cygwin from the CD directly (you will need about 200MB of disk space for the base distribution).

- Start the `cygwin\setup.exe` program.
- It will prompt you for whether you want to install from the Internet or from the local directory. Select the local directory.
- You will then be prompted for the location of the install. By default, this will also be the place where your home directory is. You don't need to install on the `C:` drive necessarily, so choose a place with enough space for your work.
- On the next screen, you will be asked what you want to install. Click on the circular arrow next to `All` and *wait* for it to change to `Install`. We will be installing everything. If you do not want to, make sure at least you install `X11`, `sh`, `gcc`, and `rpm`. You may want to install `cvs` and `ssh` as well.
- Select `done`, and Cygwin will start to install. This takes a while, usually about 15min or so.

1.2.2 Customizing your environment

Once you have installed Cygwin, you will have a raw environment, and you need to take a few steps to make it usable.

- First, you may want to set some environment variables in Windows. You set them in Control Panel-System-Advanced-Set Environment Variables.
  - Set the `HOME` variable to the location of your home directory, for example `c:\cygwin\home\username`.
  - Add cygwin to your path by appending the text `;c:\cygwin\bin;c:\cygwin\X11R6\bin` to the PATH variable.
- If you start Cygwin, you will get a bash shell. We'll be doing most of our work in X, so the first thing you want to do is create a `.xinitrc` file containing a line that starts a window manager. The `fvwm` window manager is a good choice.
- Once you have the `.xinitrc` file created, start X by running the command `xinit` at the shell prompt. You will get an X session that covers the entire screen. From here, you should start a terminal window. This may be done by selecting a shell/command prompt using the left or right mouse button.

1.2.3 Installing the remaining tools

Installing OCaml, OMake, and MetaPRL in Cygwin is similar to the process in Linux. If you installed `rpm`, then you should install the RPM packages for OCaml and OMake, and copy the MetaPRL distribution to your home directory.

1.2.4 Building MetaPRL

Once you have edited the `mk/config` you should be able to run `omake` and have it build everything. Run the following command in the MetaPRL directory.

...metaprl> omake
...lots of messages...

1.3 Configuring MetaPRL

We will need to configure the MetaPRL build-system to recognize the new theory we are creating. Normally, logical theories are placed in separate directories within the subdirectory *MetaPRL-main-directory*`/theories`. Table 1 shows a listing of the standard distribution. Each of the theory directories contains a definition of a logic, or support for a logic.

base base logic, including basic display forms for modules, and base resources for proof automation,

Table 1: Listing of the `theories` directory

```
% cd ~/metaprl/theories
% ls
CVS   experimental  itt     ocaml_doc  sil
base  fir           lf      ocaml_sos  tptp
czf   fol           mojave  phobos     tutorial
```

czf Aczel's CZF set theory, including its embedding into type theory,

fol a complete definition of first-order logic,

itt the Nuprl-like computational type theory,

lf the Edinburgh Logical Framework (incomplete),

mojave an experimental compiler based on formal rewriting in MetaPRL

ocaml_doc an introduction into OCaml programming language

ocaml_sos an operational semantics for OCaml programs,

phobos a theory used by an extensible parser Phobos,

sil the definition and semantics of a "simple imperative language,"

tptp an interface to the TPTP set of problems for theorem provers,

tutorial the directory for this tutorial

Once ocaml and omake are installed, you should be able to run `omake` in the top-level MetaPRL directory. The first time you run `omake`, it will create a new `mk/config` file that you will need to edit.

...metaprl> omake
...messages...
A new config file mk/config was created for you.
You should edit it before continuing.
*** omake: include file error: mk/config

We will use the `tutorial` directory for defining the logic in this tutorial. To add this directory to the build system, the file `mk/config` must be modified to include the `tutorial` directory in the build. This is specified by the `THEORIES` variable, which should be modified to include the `tutorial` directory as follows.

```
THEORIES=base itt tutorial
```

The `base` theory is always included in the build; we include the `itt` theory in the build because we will be relating our first-order logic to the PRL computational type theory, and we also include the `tutorial` directory.

Table 2: Boilerplate `OMakefile` code

```
OCAMLINCLUDES +=\
        $(addprefix -I , $(SUPPORT_DIRS))\
        -I ../base -I ../itt

# Library files
MPFILES =

Theory($(MPFILES))

#
# Clean up
#
clean:
    $(CLEAN)

all: theory$(LIB_SUFFIX)
```

MetaPRL uses the `omake` build the system, so we will need to create an `OMakefile` in the `tutorial` directory. This file is boilerplate, and all theories have the basic `OMakefile` contents shown in Table 2. As files are added to the theory, they must also be added to the `MPFILES` variable to be included in the build (at the moment there are no files in the theory yet).

Once the `mk/config` file has been edited, and the `theories/tutorial/OMakefile` created, MetaPRL can be built by using the `omake` command, as follows:

```
% cd ~/metaprl
% omake
...compilation messages...
```

2 Introduction

In this tutorial we develop an account of intuitionistic and classical first-order logic, and we relate it to the Nuprl type theory `Itt_theory`. We develop first-order logic, not because it is ideally suited to MetaPRL (MetaPRL is designed for higher-order logics), but because it serves as an excellent example showing how to define a logic and relate it to others in the system.

The theory we develop is fully modular. It contains modules for each of the logical operators, and the constructive logic is a subtheory of the classical logic. We also investigate proof automation. Each module defines proof procedures for its operator, and the final logic defines an automated proof procedure for the logic.

The first step in creating a logic is creating a module for each of the logical operators. Initially, the framework has no notion of truth or falsehood in the logic; these concepts are defined by declaring syntax and logical rules. Our logic will include the logical operators in first-order logic (`FOL`), and it will include terms to represent *true* and *false*. We will use a single-conclusion sequent calculus (a multi-conclusion sequent calculus is also possible, but this choice will make it easier to relate the logic to the type theory).

This presentation is divided into several parts. In Section 3, we develop a primitive account of the constructive part of first-order logic; then in Section 4 we define an interpretation in the Nuprl type theory. The constructive logic is extended to a classical first-order logic in Section 5. Finally, in Section 6 we build a model of classical first-order logic in the Nuprl type theory.

3 Defining a First-Order Logic

The presentation takes the form of a tutorial, and we will explicitly go through all the steps necessary to define the first-order logic and relate it to the type theory.

3.1 Logical constants

The first step will be to create modules for the logical constants *true* and *false*. Our logic will eventually have several modules, one for each of the logical operators. Each module has an interface defined in a `.mli` file, and an implementation defined in a `.ml` file. The interface defines the syntax, and the implementation provides the rules, display, and tactics that implement the module.

3.1.1 The `False` interface

First, define the interface for the false operator in the module `Fol_false`.

• *Create the file* `fol_false.mli`*, containing the line:*

```
extends Base_theory
declare "false"
```

The declaration defines a term `"false"` to represent the constant *false*.

3.1.2 The `False` implementation

The implementation for the `false` constant contains the same syntax declaration as in the interface. It also includes a display form, and it defines the basic rules for falsehood.

• *Create the file* `fol_false.ml`*, containing the following lines:*

```
extends Base_theory

declare "false"

dform false_df : "false" = ‘"false"

prim false_elim ’H :
 sequent { <H>; "false"; <J> >- ’e } =
 trivial
```

The `extends` directive establishes the dependency of the `Fol_false` module on the `Base_theory` module. This module is included for basic logic and display form definitions. The display form defines the print representation of the term we have just defined — the `false` term is displayed as the string literal `"false"`. The inference rule, `false_elim`, states that anything can be derived from a `false` assumption. The rule is declared as primitive, since it is an *axiom* in the first-order logic. There is no computational content to the rule, and the proof term is the trivial proof `trivial`.

The next step is to add the new module to the build system.

• *Edit the* `OMakefile`*, and add the following line:*

```
MPFILES = fol_false
```

• *Run the* `omake all` *command*

```
% omake all
...compilation messages...
```

This step creates several new files.

fol_false.cmi is the compiled interface file,

fol_false.cmo is the compiled implementation file,

fol_false.ppo is a version of the implementation file that has been preprocessed by the MetaPRL compiler,

fol_false.cmiz summarizes the logical content of the interface file `fol_false.mli`,

fol_false.cmoz summarizes the logical content of the implementation file `fol_false.ml`.

Table 3: Editor listing for the `fol_false` module

```
% cd editor/ml
% ./mpxterm
MetaPRL 0.9.5:
     build [Sun Sep 7 8:10:06 PDT 2003]
     on mojave.cs.caltech.edu
     Uses VERBOSE Refiner_ds
# cd "fol_false";;
/fol_false : string
# ls "";;
Implementation:
begin
    extends base_theory
    declare Fol_false!false (displayed as false)
end
```

The implementation is compared with the interface during the compilation, and any discrepancies in the definitions will produce an error.

Now that the module is defined, we can view it with the module editor. The MetaPRL system must be built first, and this can be performed by running `omake` (*not* `omake all`) in the MetaPRL root directory.

• *Compile the new editor, with the following commands:*

```
% cd ~/metaprl
% omake
...compilation messages...
```

• *Start the editor, and view the* `Fol_false` *module:*

The theory listing is shown in Table 3. It includes the declaration with its fully qualified name `Fol_false!false` and its display form. The editor can be exited with the command `exit ();;`, or with EOF (usually `^D` in Emacs).

3.1.3 The `True` constant

The module defining the constant `true` is similar to the `false` module. The `Fol_true` module declares a new term for `true`, and it also defines the computational content of `true` as the proof term `trivial`. The contents of the `fol_true` module is shown below.

```
Interface:
declare "true"
```

Implementation:

```
extends Base_theory

declare "true"

dform true_df : "true" = `"true"

prim true_intro :
   sequent { <H> >- "true" } = trivial
```

3.2 Basic Automation

When a rule is declared, the MetaPRL compiler produces a *tactic* with the same name that can be used to apply the rule during a proof. The `true_intro` and `false_elim` rules have the following declarations:

```
# true_intro;;
- : tactic
# false_elim;;
- : int -> tactic
```

A *tactic* is a function that applies the rule by backward-chaining and matching. For the `false_elim` rule, the second-order variable `'H` represents an arbitrary list of hypotheses, and the tactic for `false_elim` requires an explicit *int* argument that specifies the index of the `false` hypothesis. In general, the int argument corresponding to `'H` specifies how many hypotheses in the actual goal sequent match the rule's second order variable H, incremented by 1.

The use of raw rules can be rather unwieldy. In each step of a proof, a particular rule must be selected for application, and its parameters have to be constructed explicitly. If a logic has more than a few rules, this process can be extremely time-consuming. The obvious solution is to provide proof search procedures that 1) automate the proof search process, and 2) categorize the *styles* of reasoning.

A simple example is helpful to describe this approach. In sequent calculi, rules are often classified as "introduction" and "elimination" rules. In a goal-directed proof, introduction rules decompose logical operators to the right of the sequent turnstile, and elimination rules decompose logical operators on the left. As we develop our first-order logic, we will need to provide elimination and introduction rules for each of the operators that we add—for conjunction, disjunction, implication, etc. Then, if we were to perform a proof, say of the sequent $A \Rightarrow B \Rightarrow C \vdash (A \wedge B) \Rightarrow C$, we would use a sequence of introduction and elimination rules.

$A \Rightarrow B \Rightarrow C \vdash (A \wedge B) \Rightarrow C$
BY implies_intro 1
$A \Rightarrow B \Rightarrow C,\ A \wedge B \vdash C$
BY and_elim 2
$A \Rightarrow B \Rightarrow C,\ A,\ B \vdash C$
...more...

This process is tiresome because in each step we have to remember the name of the rule to apply, and figure out its arguments. But intuitively, the sequence of steps is just a sequence of "decompositions" of logical operators. It would be much easier to have a *single* tactic that is able to perform general decompositions given the address of the term to decompose. As we define elimination and introduction rules, we would then add their descriptions to this generic tactic. A tactic like this is *context-sensitive:* its behavior depends on the set of rules that are within scope when the tactic is applied.

MetaPRL implements a general mechanism, called a *resource*, for defining context-sensitive tactics (and other kinds of functions). A resource has a name, and it is declared with four types:

```
resource (insert_type, result_type) name
```

A resource can be thought of abstractly as a collection of values of *insert_type*. A resource has two basic operations: *insertion* adds a new component of type *insert_type* to a resource, and *extraction* produces a value of type *result_type* from the resource. The main difference between a resource and a traditional collection like a list is that MetaPRL maintains scoping behavior for resources. The value of a resource in a proof includes all insertions from extended modules, and all insertions in the current module up to the item being proven.

Since sequent calculi are so common, the MetaPRL base theory defines resources for collecting introduction and elimination rules, and using them to build a generic "decomposition" tactic `dT`. The definition of these resources are as follows:

```
resource (term * (int -> tactic), int -> tactic) elim
resource (term * (string * int option * tactic), tactic) intro
```

Insertion into these resources requires a term argument that defines a pattern that is being decomposed as well as a tactic that performs the decomposition. The elimination resource also requires an integer argument that identifies the hypothesis that is being decomposed.

The `dT` tactic consults these resources when "decomposing" logical operators. It is declared as follows:

```
val dT : int -> tactic
```

The argument to `dT` is the clause number that the tactic is applied to: if the argument is 0, the tactic applies the introduction rule to the conclusion (`dT` is defined only for single-conclusion sequent calculi). Otherwise, the clause number specifies the hypothesis number for application of an elimination rule.

3.2.1 Manual addition to the dT tactic

We can add the introduction and elimination rules to the dT tactic by adding proof procedures to the intro and elim resources. To illustrate the process, we will add proof automation to the fol_true and fol_false modules.

• *Add the introduction rule to the* intro *resource by adding the following line after* true_intro *function:*

```
let resource intro +=
  [<< "true">>, ("true_intro", None, true_intro)]
```

This definition adds the true_intro tactic to the intro resource to be used when decomposing a term of the form "true". The update is functional; it returns a new copy of the resource where the insertion has been performed.

The same sequence of steps can be used to add the false_elim rule to the elim resource.

• *Add the* false_elim *tactic to the* elim *resource by adding the following lines after the* false_elim *function:*

```
let resource elim +=
   [<< "false" >>, false_elim]
```

At this point, the automation is complete (at least for adding the rules for false to the dT tactic), but the proofs we can perform with just the terms true and false are rather limited. We will revisit proof automation after we have added more operators to the theory.

3.2.2 Automatic addition of automation

If we continue adding automation to other modules, we will notice that the steps are almost identical to adding automation to the fol_true and fol_false modules: after we define the rule, we would add that rule and the conclusion or hypothesis term the rule operates on to the appropriate resource. These steps are all boilerplate code, and adding the proof automation manually results in much the same code in almost all situations.

To address this problem, MetaPRL has an alternative method for automatically adding rules to resources as the rules are defined. Automatic insertion into resources is performed by adding *annotations* to the rule definitions. The annotation is enclosed in $\{|\cdots|\}$ brackets just after the *name* of the rule and before any extra argument. The annotation includes the name of a resource and any additional arguments.

• *Add resource annotations to the rules in the* Fol_true *module, by modifying the rule definitions as follows:*

```
open Dtactic

prim true_intro {| intro [] |} :
   sequent { <H> >- "true" } = trivial
```

These rule annotations add the rules directly to the `intro_resource`. Essentially, the MetaPRL compiler constructs a version of the code that we added manually before. We need to open the Dtactic module in order to access the ML function defined there that knows how to interpret the annotation, automatically figuring out the appropriate values to pass to the resource. The `[ ]` brackets are the empty list of options passed to the `intro_resource`; we don't need to pass any extra arguments here, and this list is empty.

3.3 Adding the propositional connectives

The next step in the development of the first-order logic is to define the propositional connectives for conjunction, disjunction, implication, and negation. In this section, we will show how to add the definition of implication; the modules for conjunction and disjunction are similar. Negation can be derived from implication and the constant `"false"`, as we show in the next section.

The interface for implication follows the standard format. We extend the `Base_theory` module and we declare a new binary operator for `implies`. We also define proof terms `lambda` and `apply`, and display form precedences for each of the new terms.

• *Define the* `Fol_implies` *interface by adding the following lines to the file* `fol_implies.mli`*:*

```
extends Base_theory

prec prec_implies
prec prec_lambda
prec prec_apply

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

The implementation for the `implies` term includes the declarations of the interface, and it also defines a display form, and the introduction and elimination rules for implication.

• *Declare the syntax for the* `implies` *operator by adding the following lines to the file* `fol_implies.ml`*:*

```
extends Base_theory

open Dtactic

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

The next step is to provide display forms that print these terms in their more standard representations. The display forms for these terms require parenthesization for non-ambiguous parsing, so we introduce several precedence declarations.

• *Provide precedence declarations for pretty-printing, by adding the following lines to the* `fol_implies.ml` *file:*

```
prec prec_implies
prec prec_lambda
prec prec_apply

prec prec_lambda < prec_apply
prec prec_lambda < prec_implies
prec prec_implies < prec_apply
```

These precedence declarations require a partial order on the display precedences

`prec_lambda < prec_implies < prec_apply`.

The display forms are added with `dform` declarations.

• *Add display forms for each of the terms in* `Fol_implies`*:*

```
dform implies_df : parens ::
   "prec"["prec_implies"] :: implies{'A; 'B} =
   szone pushm[0] slot["le"]{'A} hspace
      Rightarrow `" " slot{'B} popm ezone
dform lambda_df : parens ::
   "prec"["prec_lambda"] :: lambda{x. 'b} =
   szone pushm[3] Nuprl_font!lambda slot{'x} `"."
      slot{'b} popm ezone
dform apply_df : parens ::
   "prec"["prec_apply"] :: apply{'f; 'a} =
   slot{'f} hspace slot{'a}
```

There are several parts to these display form definitions. Display forms contain *zones* that determine the behavior of line-breaking within the zone. A *soft* zone (defined with the term `szone`) has the behavior that either 1) *all* hard-breaks are taken (the term `hspace` is a hard-break that either prints a space character or causes a line break), or 2) *none* of them are. The display forms also contain indentation zones, defined with the terms `pushm` and `popm`. A `pushm`[i] term indents all lines up until the next `popm` term to the current column plus i characters.

The `slot` terms are used to display subterms and perform parenthesization. In the `implies_df` display form, the `slot{'A}` term displays the subterm represented by `'A`, wrapping the displayed representation of `'A` in parentheses if the precedence of the display form for `'A` is *less* than `prec_implies`. The term `slot["le"]{'A}` displays `'A` with parentheses if its precedence is *no more than* `prec_implies`.

The `Rightarrow` term is defined in the module `Nuprl_font`, and it displays as the character ⇒. These definitions result in the following kinds of output.

Term	Display
`implies{true; true}`	$True \Rightarrow True$
`implies{implies{'A; 'B}; 'C}`	$(A \Rightarrow B) \Rightarrow C$
`implies{'A; implies{'B; 'C}}`	$A \Rightarrow B \Rightarrow C$
`implies{'A; apply{'B; 'C}}`	$A \Rightarrow (B\ C)$

The `Fol_implies` module defines computation over the proof terms `lambda` and `apply`. Computation uses the standard beta-reduction form.

• *Define computation over proof terms with the following rewrite rule:*

```
prim_rw beta : apply{lambda{x. 'b['x]}; 'a} <--> 'b['a]
```

This rewrite states that the beta-redex $(\lambda x.b[x])(a)$ is computationally *equivalent* to the term $b[a]$, where a has been substituted for x in $b[x]$.

The implication has introduction and elimination rules corresponding to natural deduction rules for implication: in order to prove $(A \Rightarrow B)$, assume A and prove B; if A is provable with assumption $(A \Rightarrow B)$, then B is also a valid assumption. We add the inference rules as follows.

• *Add the inference rules for implication:*

```
prim implies_intro {| intro [] |} :
   ('b['x] : sequent { <H>; x: 'A >- 'B }) -->
   sequent { <H> >- 'A => 'B }
   = lambda{x. 'b['x]}
prim implies_elim {| elim [] |} 'H :
   ('a : sequent { <H>; 'A => 'B; <J> >- 'A }) -->
   ('x['b] : sequent
     { <H>; f: 'A => 'B; <J>; b: 'B >- 'C }) -->
   sequent
     { <H>; f: 'A => 'B; <J> >- 'C } = 'x['f 'a]
```

The proof terms for the rules use second-order binding. For instance, the proof term for the `implies_intro` rule states that if the computational content of the first clause is `'b['x]`—that it is a term with a free variable x—then the computational content of the goal clause is $\lambda(x.b[x])$.

The modules for conjunction and disjunction are similar to implication. These modules are shown in tables 4, 5, and 6. The module for conjunction represents the computational content of the conjunction $A \wedge B$ as the pair (a, b) where a is the content of the proposition A, and b is the content of the proposition B. The `spread` term is the `pair` destructor.

The module for disjunction is similar: the computational content of a disjunction $A \vee B$ is either the term $inl(a)$ or $inr(b)$ where a is the computational content of A, and b is the computational content of B. The `decide` term is the corresponding destructor. The disjunction has *two* introduction rules, one for each branch of the disjunction. This is a special case where the `intro_resource` requires an argument to distinguish selection of the rule to apply during disjunction decomposition. The option `SelectOption` is defined in the `Dtactic` module, and it means that the `dT` tactic requires a "selection"

Table 4: `Fol_and` module for conjunction

```
extends Base_theory

open Dtactic

declare "and"{'A; 'B}
declare "pair"{'a; 'b}
declare spread{'x; a, b. 'body['a; 'b]}

prec prec_and

dform and_df : parens :: "prec"["prec_and"] ::
   "and"{'A; 'B} =
   szone pushm[0] slot{'A} hspace wedge
   `" " slot{'B} popm ezone

dform pair_df : "pair"{'a; 'b} =
   `"<" slot{'a} `"," slot{'b} `">"

dform spread_df : "spread"{'x; a, b. 'body} =
   szone pushm[0] `"let <" slot{'a} `"," slot{'b}
   `"> =" hspace slot{'x} hspace `"in" hspace
   slot{'body} popm ezone

prim_rw reduce_spread :
   spread{pair{'x; 'y}; a, b. 'body['a; 'b]} <-->
     'body['x; 'y]

prim and_intro {| intro [] |} :
   [main] ('a : sequent { <H> >- 'A }) -->
   [main] ('b : sequent { <H> >- 'B }) -->
   sequent { <H> >- 'A & 'B } = pair{'a; 'b}

prim and_elim {| elim [] |} 'H :
   [main] ('body['y; 'z] : sequent
    { <H>; y: 'A; z: 'B; <J> >- 'C }) -->
   sequent { <H>; x: 'A & 'B; <J> >- 'C }
   = spread{'x; y, z. 'body['y; 'z]}
```

Table 5: `Fol_or` module for disjunction (part 1)

```
extends Base_theory
open Dtactic

declare "or"{'A; 'B}
declare inl{'a}
declare inr{'b}
declare decide{'x; y. 'body1['y]; z. 'body2['z]}

prec prec_or
prec prec_inl
prec prec_inr
prec prec_decide

dform or_df : parens :: "prec"["prec_or"] ::
   "or"{'A; 'B} =
   szone pushm[0] slot{'A} hspace
   vee `" " slot{'B} popm ezone

dform inl_df : parens :: "prec"["prec_inl"] ::
   inl{'x} = `"inl " slot{'x}

dform inr_df : parens :: "prec"["prec_inl"] ::
   inr{'x} = `"inr " slot{'x}

dform decide_df : parens :: "prec"["prec_decide"] ::
   decide{'x; y. 'body1; z. 'body2} =
   szone pushm[3] `"match " slot{'x} `" with" hspace
   `"inl " slot{'y} `" ->" hspace slot{'body1}
   hspace `"| inr " slot{'z} `" ->" hspace slot{'body2}
   popm ezone

prim_rw reduce_decide_inl :
   decide{inl{'x}; y. 'body1['y]; z. 'body2['z]} <-->
   'body1['x]
prim_rw reduce_decide_inr :
   decide{inr{'x}; y. 'body1['y]; z. 'body2['z]} <-->
   'body2['x]
```

Table 6: `Fol_or` module for disjunction (part 2)

```
(* Rules *)
prim or_intro_left
   {| intro [SelectOption 1] |} :
   [main] ('a : sequent { <H> >- 'A }) -->
   sequent { <H> >- "or"{'A; 'B} } =
   inl{'a}

prim or_intro_right
   {| intro [SelectOption 2] |} :
   [main] ('b : sequent { <H> >- 'B } ) -->
   sequent { <H> >- "or"{'A; 'B} } =
   inr{'b}

prim or_elim {| elim [] |} 'H :
   ('a['x] : sequent
      { <H>; x: 'A; <J> >- 'C }) -->
   ('b['x] : sequent
      { <H>; x: 'B; <J> >- 'C }) -->
   sequent
      { <H>; x: 'A or 'B; <J> >- 'C } =
   decide{'x; x. 'a['x]; x. 'b['x]}
```

argument (either 1 or 2 in this case) that identifies the rule to apply during decomposition. The correct usage of the `dT` tactic for disjunction-introduction requires the use of the `selT : int -> tactic -> tactic` *tactical* to provide the selection argument. Valid uses are: `selT 1 (dT 0)` and `selT 2 (dT 0)`.

3.4 Defining negation

Negation is a special case of propositional operator because it can be defined in terms of implication and falsehood, with the definition $\neg A \equiv A \Rightarrow \mathit{False}$. It is possible to define the introduction and elimination rules for negation directly, but it is better to derive the rules from the rules for negation and falsehood, rather than defining them as primitive.

The interface for negation declares the syntax as usual, but it establishes a dependency on the `Fol_false` and `Fol_implies` modules, and it declares the negation definition.

• *Declare the interface for negation by adding these lines to the file* `fol_not.mli`*:*

```
extends Fol_implies
extends Fol_false

declare "not"{'A}

rewrite unfold_not : "not"{'A} <-->
   implies{'A; ."false"}

prec prec_not
```

The `extends` directives establish the logical module dependencies. The `declare` defines the syntax for negation, and the `rewrite` term establishes the definition of `not` as a computational equivalence. Computational rewriting is the general definitional mechanism in MetaPRL: this definition states that, in all contexts, the term `not{'A}` is equivalent to the term `implies{'A; false}`.

In the implementation file, the definitional nature of this equivalence will be made explicit.

• *Define the logical dependencies* `fol_not.ml` *file:*

```
extends Fol_false
extends Fol_implies

open Dtactic
```

• *Provide the definition of negation as implication of falsehood:*

```
define unfold_not : "not"{'A} <-->
   implies{'A; ."false"}
```

The `define` directive both declares the syntax for the `not`, as well as postulates a definitional equivalence (MetaPRL will make sure that `not` is not already defined). Note that this declaration in the implementation states that equivalence is definitional, where the interface simply declared the rewrite with the `rewrite` directive.

The display form for negation has the standard format. We include a precedence declaration for parenthesization.

• *Define the display properties of negation:*

```
prec prec_not
dform not_df : parens :: "prec"["prec_not"] ::
   "not"{'A} = tneg slot{'A}
```

The rules for negation include the standard introduction and elimination rules, but these rules can be derived from the rules for `false` and `implies`. In the `fol_not.ml` file, we declare the rules as *interactive*, which states that the rules are presumed valid with a proof obligation to show that they can be derived.

• *Define the derived inference rules for negation:*

```
interactive not_intro {| intro [] |} :
 sequent { <H>; 'A >- "false" } -->
 sequent { <H> >- "not"{'A} }
interactive not_elim {| elim [] |} 'H :
 sequent { <H>; "not"{'A}; <J> >- 'A } -->
 sequent { <H>; "not"{'A}; <J> >- 'C }
```

The next step is to *prove* these inference rules, which requires that we use the interactive proof editor. During this proof, we will be making use of the `dT` tactic to apply the introduction and elimination rules for `false` and `implies`. To start the proof, we need to compile a build of the system. The `MPFILES` variable in the `OMakefile` should contain a list of all the modules in the tutorial so far:

```
MPFILES = fol_false fol_true\
   fol_implies fol_and fol_or fol_not
```

• *Build a version of MetaPRL that contains all the modules in the tutorial:*

```
~/metaprl% omake
...messages from compiler...
```

• *Start the proof editor, and list the* `Fol_not` *module (Table 7, Step 1).*

This listing, shown in Table 7, shows that there are two incomplete proof obligations: the rules for `not_intro`, and `not_elim`. Note the $A{<}\,|\Gamma|\,{>}[]$ syntax in the `not_elim` rule — it specifies that the second-order variable A can only have free occurrences of variables introduced by the context Γ, but not of those introduced by the context Δ, even despite the fact that this particular instance of A is in the scope of Δ. We did not have to specify this constraint

Table 7: Listing of the `Fol_not` module

Step 1

```
~/metaprl% cd editor/ml
~/metaprl/editor/ml% ./mpxterm
# cd "/fol_not";;
  /fol_not : string
# ls "";;
```

Implementation:
begin
  extends fol_implies
  extends fol_false
  declare Fol_not!not{A} (displayed as $\neg A$)
  ![] **rewrite** unfold_not : $(\neg A) \longleftrightarrow (A \Rightarrow \text{False})$
  # **rule** not_intro Γ:
        $< \Gamma >$; $A \vdash$ false $\longrightarrow$
        $< \Gamma > \vdash \neg A$
  # **rule** not_elim Γ:
        $< \Gamma >$; $\neg A$; $< \Delta > \vdash A <$ $|\Gamma|$ $>[] \longrightarrow$
        $< \Gamma >$; $\neg A$; $< \Delta > \vdash C$
end

explicitly in the `.ml` file — since one of the instances of A is outside of the scope of Δ, MetaPRL was able to figure it out automatically.

We will walk through a proof of the two rules.

• *Change the current directory to the* `not_intro` *rule (Table 8, Step 2).*

To prove this goal, we first unfold the negation using the definition `unfold_not`. The `unfold_not` definition is a *computational rewrite*, also called a *conversion*. Rewrites are like tactics: there are *conversionals* to combine rewrites and perform search; these conversionals are defined in the module `Tactic_type.Conversionals`. For our purposes in applying the `unfold_not` rewrite, we can use the `rwh` ("rewrite higher") that searches for the outermost occurrences of terms to rewrite. The `rwh` function is declared as `rwh : conv -> int -> tactic`: it takes a `conv` (a rewrite), and a clause number, and produces a `tactic` that applies the rewrite to that clause, rewriting the outermost terms.

• *Unfold the definition of the negation (Table 8, Step 3).*

The unfolding produced one subgoal where the definition of negation has been unfolded. We can descend into the subgoal using the `down` proof-navigation function, then apply the `dT` tactic.

• *Navigate to the subgoal and apply the* `dT 0` *tactic (Table 9, Step 4).*

This proof step produces two subgoals: one to prove that `'A` is a type, and another to prove that `false` is a type. For the first goal, we use the typehood assumption, using that tactic `nthAssumT`.

Table 8: Proving the `not_intro` rule (steps 2, 3)

Step 2

```
# cd "not_intro";;
```

```
/fol_not/not_intro : string
```

```
# ls "";;
```

```
#
....main....
1.  < Γ > ; A ⊢ false
=====
 < Γ > ⊢ (¬A)
BY <goal>
```

Step 3

```
# refine rwh unfold_not 0;;
```

```
#
....main....
1.  < Γ >  ; A ⊢ false
=====
 < Γ > ⊢ (¬A)
BY rwh unfold_not 0
1. [#]
    ....main....
     < Γ > ⊢ (A ⇒ false)
```

Table 9: Proving the `not_intro` rule (steps 4, 5)

Step 4

```
# down 1;;
```

```
# #
....main....
1.  < Γ >; A ⊢ false
=====
 < Γ > ⊢ (A ⇒ false)
BY <goal>
```

```
# refine dT 0;;
```

```
# #
....main....
1.  < Γ >; A ⊢ false
=====
 < Γ > ⊢ (A ⇒ false)
BY dT 0
1. [#]
    ....main....
     < Γ >; A⊢ false
```

Step 5

```
# down 1;;
```

```
# # #
....wf....
1.  < Γ >; A ⊢ false
=====
 < Γ >; A ⊢ false
BY <goal>
```

```
# refine nthAssumT 1;;
```

```
  *
....wf....
1.  < Γ >; A ⊢ false
=====
 < Γ >; A ⊢ false
BY nthAssumT 1
```

Table 10: Proof of the `not_intro` rule

```
# cd "/fol_not/not_intro";;
  /fol_not/not_intro : string
# refine rwh unfold_not 0 thenT autoT;;
  *
  ....main....
  1.  < Γ > ; A⊢ false
  =====
   < Γ > ⊢ ¬A
  BY rwh unfold_not 0 thenT autoT
  =====
  1.*
     ....main....
      < Γ > ⊢ A ⇒ false
```

• *Navigate to the subgoal, and apply the* `nthAssumT 1` *tactic (Table 9, Step 5).*

Note that the application of the nthAssumT tactic produced a `*` on the status line, meaning that this branch of the proof is complete.

At this point, all proof obligations have been satisfied, and the proof is complete.

This was a pretty simple proof, involving only the use of the `dT` and `nthAssumT` tactics. The `autoT` tactic performs "automated" proving based on repeated application of several "basic" tactics including `dT` and `nthAssumT`. We can navigate back up the proof tree, and prove the goal in one step.

• *Navigate back to the root, and apply the* `autoT` *tactic (Table 10, Auto).*

The `autoT` tactic proves the goal, and it preserved the previous subtree (the root is listed under the `BY autoT` line). The infix function `thenT` is a *tactical* used for sequencing: the proof first unfolds the `not` term, and *then* applies the `autoT` tactic.

The `not_elim` rule is the final piece of reasoning that we have to perform. The `autoT` tactic does not completely prove this goal because it requires *folding* the definition of `not`. The root of the proof tree can be expanded using the `autoT` tactic, which leaves a partial proof.

• *Expand the initial part of the* `not_elim` *proof (Table 11, Step 1).*

The second goal has a trivial proof.

• *Prove the second subgoal by using the* `dT 4` *tactic (Table 12).*

The first goal is more difficult. The goal uses the hypothesis $A \Rightarrow$ *false*, and the assumption uses the hypothesis $\neg A$. MetaPRL allows rewriting on an assumption using the `rwch` rewriting function. The `rwch` function takes two numbers: the first number is the number of an assumption, and the second number is the hypothesis to rewrite.

• *Prove the first subgoal using the* `rwch` *rewriting function (Table 13).*

Table 11: Prove the `not_elim` rule (step 1)

Step 1

```
# cd "../not_elim";;
```

/fol_not/not_elim : string

```
# ls "";;
```

#
....main....
1. $< \Gamma >$; $\neg A$; $< \Delta >\vdash A< |\Gamma| >[]$
=====
$< \Gamma >$; $\neg A$; $< \Delta >\vdash C$
BY <goal>
() : unit

```
# refine rwh unfold_not 2 thenT dT 2;;
```

#
....main....
1. $< \Gamma >$; $\neg A$; $< \Delta >\vdash A< |\Gamma| >[]$
=====
$< \Gamma >$; $\neg A$; $< \Delta >\vdash C$
BY rwh unfold_not 2 thenT dT 2
1. [#]
....main....
$< \Gamma >$; $A \Rightarrow$ false; $< \Delta >\vdash A< |\Gamma| >[]$
2. [#]
....main....
$< \Gamma >$; $A \Rightarrow$ false; $< \Delta >$; false$\vdash C$

Table 12: Prove the `not_elim` rule (step 2)

Step 2

```
# down 2;;
```

```
# #
....main....
1.  < Γ > ; ¬A; < Δ >⊢ A< |Γ| >[]
====
 < Γ > ; A ⇒ false; < Δ >; false⊢ C
BY <goal>
() : unit
```

```
# refine dT 4;;
```

```
# *
....main....
1.  < Γ > ; ¬A; < Δ >⊢ A< |Γ| >[]
====
 < Γ > ; A ⇒ false; < Δ >; false⊢ C
BY dT 4
```

Table 13: Prove the `not_elim` rule (step 3)

Step 3

```
# cd "../1";;
```

```
/fol_not/not_elim/1 : string
```

```
# refine rwch unfold_not 1 2 thenT nthAssumT 1;;
```

```
* *
....main....
1.  < Γ > ; ¬A; < Δ >⊢ A< |Γ| >[]
====
 < Γ > ; A ⇒ false; < Δ >⊢ A< |Γ| >[]

BY rwch unfold_not 1 2 thenT nthAssumT 1
```

Table 14: Finish the `fol_not` module

```
# cd "~";;
  /fol_not : string
# ls "u";;
  Implementation:
  begin
  end
  () : unit
# save ();;
  () : unit
```

This completes the derivation. As a final step, we can change back to the module directory and check that all obligations have been satisfied. The `ls "u"` command lists only the unproved rules in a module. After listing, we can *save* the theory to the logical library.

• *Check that all obligations are satisfied and save the module (Table 14).*

3.5 Adding structural rules

The logical primitives defined so far—`false`, `true`, `and`, `or`, `implies`, and `not`—together define a *constructive* propositional logic, but they are lacking some common *structural* rules. Structural rules define operations on *sequents* without an explicit connection to any particular logical connective. We need three structural rules: one for a proof by *assumption*, another for *thinning* of hypotheses, and another for *cut* (addition of a lemma).

We will define these structural rules in a module called `Fol_struct`. The contents of the `fol_struct.ml` file is shown in Table 15.

The structural rules are defined in three parts: first the `extends` and `open` statements establish the dependencies of the module. The `TacticType` module is opened for access to the `Sequent` module; the `TacticType.Tacticals` defines the `onSomeHypT` tactical used to augment the `trivialT` tactic, and the `Auto_tactic` defines the `trivialT` tactic.

The second part lists the rules themselves. The `hypothesis` judgment specifies that if any term T is both an assumption and a goal, then the sequent is trivially provable. The `thin_many` judgment is a form of monotonicity: if a goal C is provable from the assumptions in H and K, then it remains provable with any additional assumptions J. The `<|H|>` and `<|H;K|>` annotations specify where the free variables may come from — essentially they specify the restriction that variables introduced by hypotheses in J can not occur freely in K and C. The `cut` rule is used to establish a lemma, and it can be thought of as a form of *modus-ponens*: if the lemma T can be proved, and the goal C can be established with the use of the lemma, then the goal C is provable from the

Table 15: Structural rules (part 1)

```
extends Base_theory

open Tactic_type
open Tactic_type.Tacticals

open Auto_tactic

(* Structural rules *)
prim hypothesis 'H :
   sequent { <H>; x: 'T; <J> >- 'T } = 'x

prim thin_many 'H 'J :
   ('t : sequent { <H>; <K> >- 'C }) -->
   sequent { <H>; <J>; < K<|H|> > >- 'C<|H;K|> } =
   't

prim cut 'T :
   ('a : sequent { <H> >- 'T }) -->
   ('b['x] : sequent { <H>; x: 'T >- 'C }) -->
   sequent { <H> >- 'C } = 'b['a]

(* Tactics *)
let nthHypT = hypothesis

let thinAllT i j = funT (fun p ->
   let i = Sequent.get_pos_hyp_num p i in
   let j = Sequent.get_pos_hyp_num p j in
      thin_many i (j-i+2) )

let nthAssumT = argfunT (fun i p ->
   let assum = Sequent.nth_assum p i in
      Top_tacticals.thinMatchT thin_many assum thenT
      nthAssumT i)

let thinT i = thinAllT i i

let assertT = cut
```

Table 16: Structural rules (part 2)

```
(* Add "hypothesis" to the trivialT tactic *)
let resource auto += [
   {
      auto_name = "nthHypT";
      auto_prec = trivial_prec;
      auto_tac = onSomeHypT nthHypT;
      auto_type = AutoTrivial;
   } ; {
      auto_name = "Fol_struct.autoAssumT";
      auto_prec = trivial_prec;
      auto_tac = onSomeAssumT nthAssumT;
      auto_type = AutoTrivial;
   }]
```

Table 17: Tactics description for structural rules (fol_struct.mli)

```
open Refiner.Refiner.TermType
open Tactic_type.Tacticals

topval nthHypT :int -> tactic

topval thinT : int -> tactic
topval thinAllT : int -> int -> tactic

topval assertT : term -> tactic

topval nthAssumT : int -> tactic
```

original assumptions.

The `thinAllT` tactic is a wrapper around the `thin_many` rule that interprets its integer arguments as a hypotheses range, passing the correct hypotheses counts to the `thin_many` rule. The `thinT` tactic may be used when we only want to thin a single hypothesis instead of thinning a whole range. We also create an updated version of the `nthAssumT` tactic will be able to prove the goal whenever it would match an existing assumption after appropriate thinning.

The final step is to augment the standard tactics for reasoning. By convention, the `trivialT` tactic is a tactic (defined from the `AutoTrivial` improvements of the `auto` resource) that proves "trivial" goals. The definition of "trivial" is a matter of judgment, but it includes cases where the goal clause is also one of the assumptions, as in the `hypothesis` rule. The `trivialT` is defined as a component of the `autoT` tactic. The `autoT` tactic is essentially programmed with a list of tactics (in the `auto_resource` resource) that are each associated with a *precedence*. When the `autoT` tactic is applied to a goal, it applies each of the tactics in the list, from highest-to-lowest precedence until one of the tactics succeeds or all of them fail. If a tactic in the list succeeds and it produces subgoals, the `autoT` tactic repeats the process starting from the highest precedence tactic. If all tactics fail, the `autoT` tactic terminates. The highest precedence tactics are the ones with `auto_type` flag set to `AutoTrivial`, ordered according to their `auto_prec` flag. The `auto_type = AutoTrivial` tactics will be used both by `autoT` and `trivialT`. The `auto_type = AutoNormal` will be used by `autoT` after all the `AutoTrivial` are exhausted. Finally, `auto_type = AutoMustComplete` tactics will be used by `autoT` after `AutoTrivial` and `AutoNormal` ones, but any progress done with the help of `AutoMustComplete` tactics will be rolled back, unless `autoT` manages to fully finish the proof, producing no subgoals.

To add a proof procedure to the `autoT` tactic, four values must be provided: a name (in `auto_name`) that is used for printing debugging information, a `auto_type` flag with precedence (in `auto_prec`) that defines the precedence of the insertion, and the tactic itself (in `auto_tac`). The `Auto_tactic` module defines several precedences, including the `trivial_prec`. One of the tactic we add is a version of the `nthHypT` tactic wrapped by the `onSomeHypT` tactical, which searches for the first hypothesis where the `nthHypT` tactic is successful.

3.6 Defining a propositional logic

The logical primitives defined so far—`false`, `true`, `and`, `or`, `implies`, and `not`—together define a complete *constructive* propositional logic. As the next step, we can define a module for propositional logic that puts all of the logical operators together. We will call this module `Fol_prop`.

• *Define a module for (constructive) propositional logic by creating two files* `fol_prop.mli` *and* `fol_prop.ml` *that both have the following contents:*

```
extends Base_theory
extends Fol_false
extends Fol_true
extends Fol_and
extends Fol_or
extends Fol_implies
extends Fol_not
```

This module contains only `extends` statements that state that the propositional logic is the combination of all the listed logics.

Let's prove a simple theorem in the logic that uses multiple connectives:

$$((A \vee B) \Rightarrow C) \Rightarrow ((A \Rightarrow C) \wedge (B \Rightarrow C)).$$

• *State the theorem by adding the following lines to the* `fol_prop.ml` *file:*

```
interactive distrib_or :
   sequent { <H> >- (('A or 'B) => 'C) =>
      (('A => 'C) & ('B => 'C)) }
```

As mentioned previously, the `autoT` tactic is able to perform simple reasoning tasks, including repeated use of the `dT` tactic to decompose logical formulas into smaller and smaller parts. We can use `autoT` as a first step in the proof.

• *Use the* `autoT` *tactic to perform the initial steps of the proof (Table 18, Step 1).*

The `autoT` tactic has performed several steps at once. With enough experience, we can eventually figure out the expected behavior of the `autoT` tactic, but we can also *view* the steps that it took. To see what it did, we can navigate *into* the proof generated by `autoT` using the command `down 0`. There is a lot of detail and redundant information in these internal proofs, but we can eventually map the entire `autoT` subtree. The steps at each address are shown in Table 19.

Obviously, there is a lot of detail in the internal proof performed by `autoT`. However, the structure of the proof is apparent: the `autoT` tactic performs basic proof steps using the `dT 0` and `nthAssumT` tactics. The `dT` tactic in turn uses the introduction and elimination rules we defined in the modules for each of the logical operators. The *resource annotations* on the rules provided the connection to the `dT` tactic. The `trivialT` tactic performed the "trivial" operations of proving the goal from an existing assumption.

The two subgoals we get are similar, we will prove the first one. In this case the `autoT` tactic was unable to make progress because the next likely step is an elimination step, and the default implementation of `autoT` in the module `Auto_tactic` does not perform elimination reasoning. Instead, we will perform the reasoning by hand.

• *Navigate to the first subgoal, and apply the* `dT 2 thenT autoT` *tactic (Table 18, Step 2).*

The `dT 2` tactic performs elimination reasoning by producing two subgoals: one is the subgoal listed in the table, and the other is the subgoal $H,\ (A \vee B) \Rightarrow C,\ A,\ C \vdash C$, which is immediately provable by the `autoT` tactic.

Table 18: Prove the `distrib_or` theorem

Step 1
refine autoT;;

#
....main....
$< \Gamma > \vdash ((A \vee B) \Rightarrow C) \Rightarrow ((A \Rightarrow C) \wedge (B \Rightarrow C))$
BY autoT
1. [#]
....main....
$< \Gamma >$; $(A \vee B) \Rightarrow C$; $A \vdash C$
2. [#]
....main....
$< \Gamma >$; $(A \vee B) \Rightarrow C$; $B \vdash C$

Step 2
down 1;; refine dT 2 thenT autoT;;

# #
....main....
$< \Gamma >$; $(A \vee B) \Rightarrow C$; $A \vdash C$
BY $< goal >$
# #
....main....
$< \Gamma >$; $(A \vee B) \Rightarrow C$; $A \vdash C$
BY dT 2 thenT autoT
1. [#]
....main....
$< \Gamma >$; $(A \vee B) \Rightarrow C$; $A \vdash A \vee B$

Table 19: Proof steps taken by `autoT`

Address	Annotation	Goal
0	`[autoT]`	$H \vdash ((A \vee B) \Rightarrow C)$ $\Rightarrow ((A \Rightarrow C) \wedge (B \Rightarrow C))$
0/0	`<compose>`	
0/0/0	`[dT 0]`	
0/0/0/0	`[Fol_implies.implies_intro [Hyp(1)] x]`	
0/0/1	`<compose>`	$H \vdash ((A \vee B) \Rightarrow C)$ *type*
0/0/1/0	`[dT 0]`	
0/0/1/0/0	`[Fol_implies.implies_type [Hyp(1)]]`	
0/0/1/1	`<compose>`	$H \vdash ((A \vee B)$ *type*
0/0/1/1/0	`[dT 0]`	
0/0/1/1/0/0	`[Fol_or.or_type [Hyp(1)]]`	
0/0/1/1/1	`<compose>`	$H \vdash A$ *type*
0/0/1/1/1/0	`[trivialT]`	
0/0/1/1/1/0/0	`[nthAssumT 0]`	
0/0/1/1/2	`<compose>`	$H \vdash B$ *type*
0/0/1/1/2/0	`[trivialT]`	
0/0/1/1/2/0/0	`[nthAssumT 1]`	
0/0/1/2	`<compose>`	$H \vdash C$ *type*
0/0/1/2/0	`[trivialT]`	
0/0/1/2/0/0	`[nthAssumT 2]`	
0/0/2	`<compose>`	$H,\ (A \vee B) \Rightarrow C \vdash (A \Rightarrow C) \wedge (B \Rightarrow C)$
0/0/2/0	`[dT 0]`	
0/0/2/0/0	`[Fol_and.and_intro [Hyp(2)]]`	
0/0/2/1	`<compose>`	$H,\ (A \vee B) \Rightarrow C \vdash A \Rightarrow C$
0/0/2/1/1	`<identity>`	$H,\ (A \vee B) \Rightarrow C \vdash A$ *type*
0/0/2/1/2	`<identity>`	$H,\ (A \vee B) \Rightarrow C,\ x\_1{:}A \vdash C$
0/0/2/2	`<compose>`	$H,\ (A \vee B) \Rightarrow C \vdash B \Rightarrow C$
0/0/2/2/0	`[dT 0]`	
0/0/2/2/0/0	`[Fol_implies.implies_intro [Hyp(2)] x_1]`	
0/0/2/2/1	`<identity>`	$H,\ (A \vee B) \Rightarrow C \vdash B$ *type*
0/0/2/2/2	`<identity>`	$H,\ (A \vee B) \Rightarrow C,\ x\_1{:}B \vdash C$

Table 20: Prove the `distrib_or` theorem

Step 3
down 1;; refine selT 1 (dT 0) thenT autoT
* *
....main....
$< \Gamma >$; $(A \vee B) \Rightarrow C$; $A \vdash A \vee B$
BY selT 1 (dT 0) thenT autoT

The remaining subgoal is not proved by `autoT` because the application of `dT 0` requires a *selection* argument to determine the branch of the disjunction. We can perform this step by hand.

• *Navigate to the subgoal, and apply the* `selT 1 (dT 0) thenT autoT` *tactic (Table 20, Step 3).*

This complete the proof of the `distrib_or` theorem.

3.7 Proof automation for constructive propositional logic

While the proof of the `distrib_or` theorem is not difficult, it still requires far too many steps of interaction. Many of these steps are obvious, and in fact we know that the truth of statements in the propositional logic are in fact decidable. For the final step in the propositional logic, we will develop a simple (although inefficient) proof procedure for propositional sentences, using Dyckhoff's algorithm.

Dyckhoff's algorithm is based on the use of three new rules for implication elimination, shown in Table 21. The algorithm is very simple: search all proof trees in some order, using the implication rules in Table 21 instead of `implies_elim` when possible. Dyckhoff's algorithm interprets negations as implications, and we will need to unfold all the definitions of negation in the goal sequent.

We will program this proof procedure as a tactic. For the first step, we need to add `open` statements for the modules that have functions we need to use.

• *Add the following* `open` *declarations to the* `fol_prop.ml` *file:*

```
open Refiner.Refiner.Term
open Refiner.Refiner.TermOp
open Refiner.Refiner.RefineError
open Tactic_type
open Tactic_type.Tacticals
open Tactic_type.Conversionals
open Dtactic
open Auto_tactic
open Fol_not
open Fol_struct
```

Table 21: Rules for implication elimination

```
(* Refinement of implication *)
interactive imp_and_rule 'H:
   [main] sequent
      { <H>; <J>; 'C => 'D => 'B >- 'T} -->
   sequent { <H>; ('C & 'D) => 'B; <J> >- 'T}

interactive imp_or_rule 'H:
   [main] sequent
      { <H>; <J>; 'C => 'B; 'D => 'B >- 'T} -->
   sequent { <H>; ('C or 'D) => 'B; <J> >- 'T}

interactive imp_imp_rule 'H:
   [main] sequent
      { <H>; <J>; 'D => 'B >- 'T} -->
   sequent { <H>;  ('C => 'D) => 'B; <J> >- 'T}
```

The `Refiner` modules define operations on MetaPRL*terms*; the `Tactic_type` modules define basic tacticals and conversionals; the `Dtactic` defines the `dT` tactic and the `Auto_tactic` defines the `trivialT` tactic; and we also need to `Fol_not` module to access the `unfold_not` rewrite, and the `Fol_struct` modules to access the `thinT` tactic.

For the next step, we will want to state and derive Dyckhoff's rules for implication elimination.

• *Add the rules and tactics in Table 21 to the* `fol_prop.ml` *file.*

The tactics define the standard wrappers for providing arguments to the rules. The proofs of the rules are straightforward and we omit the proofs here.

The next step is to define ML functions that test whether a term matches one of the special cases of implication. The `Refiner` module provides many operations for examining terms, including accessing the opname, the parameters, and the subterms of a term.

• *Add the term predicate functions in Table 22 to the* `fol_prop.ml` *file.*

In general, operations on terms require the *opname*, which can be computed with the `opname_of_term` function. An implication has opname `implies_opname`, and it has two subterms, each with no bindings (by naming traditions, it is a `dep0_dep0` term because it has two subterms, each with 0 binding variables). The function `is_imp_and_term` $(A \wedge B) \Rightarrow C$ returns *true* because the outermost operator is an implication, and the first subterm $A \wedge B$ is a conjunction.

For the final step we write the code that performs the proof search.

• *Provide a proof search algorithm by adding the code in Table 23 to the* `fol_prop.ml` *module.*

The proof search algorithm has two parts: the `decompPropDecideHypT` applies an elimination rule, and the `decompPropDecideConclT` applies an in-

Table 22: Term predicate functions for testing implication

```
let false_opname = opname_of_term << "false" >>
let is_false_term = is_no_subterms_term false_opname

let and_opname = opname_of_term << 'A & 'B >>
let is_and_term = is_dep0_dep0_term and_opname

let or_opname = opname_of_term << 'A or 'B >>
let is_or_term = is_dep0_dep0_term or_opname

let implies_opname = opname_of_term << 'A => 'B >>
let is_implies_term =
   is_dep0_dep0_term implies_opname

let is_imp_and_term term =
   is_implies_term term
   & is_and_term (fst (two_subterms term))

let is_imp_or_term term =
   is_implies_term term
   & is_or_term (fst (two_subterms term))

let is_imp_imp_term term =
   is_implies_term term
   & is_implies_term (fst (two_subterms term))
```

Table 23: Proof search algorithm

```
(* Try to decompose a hypothesis *)
let rec decompPropDecideHypT i p =
  let term = Sequent.nth_hyp p i in
     if is_false_term term then
       dT i
     else if is_and_term term or is_or_term term then
       dT i thenT funT internalPropDecideT
     else if is_imp_and_term term then
       (* {C & D => B} => {C => D => B} *)
       imp_and_rule i thenT funT internalPropDecideT
     else if is_imp_or_term term then
       (* {C or D => B} => {(C => B) & (D => B)} *)
       imp_or_rule i thenT funT internalPropDecideT
     else if is_imp_imp_term term then
       (* {(C => D) => B} => {D => B} *)
       imp_imp_rule i thenT funT internalPropDecideT
     else if is_implies_term term then
       dT i thenT thinT i thenT funT internalPropDecideT
     else
       (* Nothing recognized, try to see if we're done. *)
       nthHypT i

(* Decompose the goal *)
and decompPropDecideConclT p =
   let goal = Sequent.concl p in
     if is_or_term goal then
       (selT 1 (dT 0) thenT funT internalPropDecideT)
       orelseT (selT 2 (dT 0)
                thenT funT internalPropDecideT)
     else if is_and_term goal or is_implies_term goal then
       dT 0 thenT funT internalPropDecideT
     else
       progressT trivialT

and internalPropDecideT p =
  onSomeHypT (fun i -> funT (decompPropDecideHypT i))
     orelseT funT decompPropDecideConclT

let propDecideT =
  onAllClausesT (fun i -> tryT (rw (sweepUpC unfold_not) i))
  thenT funT internalPropDecideT
```

Table 24: Applying the `propDecideT` tactic to `distrib_or`

....main....
$< \Gamma > \vdash ((A \vee B) \rightarrow C) \rightarrow ((A \rightarrow C) \wedge (B \rightarrow C))$
BY propDecideT

Table 25: Detailed proof status

```
# cd "fol_prop/distrib_or2";;
Module: /fol_prop
/fol_prop/distrib_or2 : string
# check();;
The proof of /fol_prop/distrib_or2 depends
on the following definitions and axioms:
Rule /fol_and/and_intro
Rule /fol_implies/implies_elim
Rule /fol_implies/implies_intro
Rule /fol_or/or_intro_left
Rule /fol_or/or_intro_right
Rule /fol_struct/hypothesis
() : unit

#  term_of_extract [];;
 < Γ > ⊢ λx. < (λv1.x(inl    v1)), (λv1.x(inr    v1)) >: term
```

troduction rule. Both functions perform a case analysis first to select the rule to apply. The product of this search code is the `propDecideT` tactic, which first unfolds all occurrences of negations in all clauses of the sequent, and then applies the internal version of the search procedure.

Let's see how this works.

• *Make a copy of the* `distrib_or` *rule statement as a new rule with name* `distrib_or2`*. Edit it, and apply the* `propDecideT` *tactic (Table 24).*

The statement is proved. Now we can ask MetaPRL to give us some information about the proof (see Table 25). The `check()` command lists all the primitive rules that the proof depends on and `term_of_extract` function can show us the computational content of the proof.

Table 26: The `Fol_pred` module

```
extends Basic_theory

(* Syntax and display *)
declare pred
dform pred_df : pred = ‘"Pred"
```

3.8 The quantifiers

To complete the definition of the first-order logic, we need to add the quantifiers $\forall$ and $\exists$. Their definitions are much like the propositional connectives with one exception: they require a binding mechanism. The informal meaning of a formula $\forall x.B[x]$ is that for any *predicate* x, the predicate $B[x]$ is true. To implement the quantifiers, we first need to provide a module `Fol_pred` that defines a term standing for all predicates.

• *Define the term for predicates with the code in Table 26.*

Predicates are, by definition, types. We also add this inference to the `trivialT` tactic.

Next, we will define a module for the universal quantifier.

• *Define the module for* `Fol_all` *with the code in Table 27.*

The `all_intro` rule uses quantifications over predicates. The `all_elim` rule uses a new option for the `elim_resource` that specifies that the hypothesis $\forall y.B[y]$ should be thinned by default using the `thinT` tactic. Otherwise, there is little that is new in these definitions. The `Fol_exists` module is similar, and we omit its definition.

As the final step in the defining the (constructive) first-order logic, we collect all the modules into a single module `Fol_theory`.

• *Define the constructive first-order logic with the code in Table 28.*

The `Fol_theory` module includes all of the modules in the first-oder logic, and it also includes *relations* between the display precedences of the display forms for each of the operators.

4 Relating the propositional logic to type theory

So far, we have developed the first-order logic as an isolated theory without any relations to any of the other logics in MetaPRL. In this section, we'll see how to relate the propositional fragment of the logic to the Nuprl type theory. The correspondence is guided by the Curry-Howard isomorphism, which gives a interpretation of each logical operator as a type constructor.

The *mechanism* for providing a derivation uses both the `extends` and `derive` statements in the MetaPRL module system. There are three basic steps in a derivation of a logical operator:

1. extend the Nuprl type theory,

Table 27: Definition of the universal quantifier

```
extends Fol_implies
extends Fol_struct
extends Fol_pred

open Dtactic
open Fol_struct

declare "all"{x. 'B['x]}

prec prec_all

dform all_df : parens :: "prec"["prec_all"] ::
   "all"{x. 'B} =
   szone pushm[3] forall slot{'x} `"."
   hspace slot{'B} popm ezone

prim all_intro {| intro [] |} 'x :
   [main] ('b['x] : sequent
      { <H>; x: pred >- 'B['x] }) -->
   sequent { <H> >- "all"{y. 'B['y]} } =
   lambda{y. 'b['y]}

prim all_elim
 {| elim [ThinOption thinT] |} 'H 'a :
 [main] ('b['x; 'z] : sequent
  { <H>; x: "all"{y. 'B['y]};
    <J>; z: 'B['a] >- 'C }) -->
 sequent { <H>; x: "all"{y. 'B['y]};
    <J> >- 'C } =
 'b['x; 'x 'a]
```

Table 28: The first-order logic `Fol_theory`

```
extends Base_theory
extends Fol_false
extends Fol_true
extends Fol_and
extends Fol_or
extends Fol_implies
extends Fol_not
extends Fol_struct
extends Fol_pred
extends Fol_all
extends Fol_exists

open Fol_implies
open Fol_and
open Fol_or
open Fol_not
open Fol_all
open Fol_exists

prec prec_implies < prec_and
prec prec_implies < prec_or
prec prec_or < prec_and
prec prec_and < prec_not
prec prec_all < prec_implies
prec prec_exists < prec_implies
```

2. state the *interpretations* of the logical operators using the `rewrite` mechanism,

3. `derive` the rules for the logical operator.

4.1 Defining well-formed formulas

Before we can derive the first-order logic from the type theory, we have to update it a little. While we were developing the first-order logic, we were implicitly assuming that any formula in the language is a well-formed proposition. Once we embed the logic into the type theory, this assumption will no longer hold.

In order to compensate for this, we declare a term to define *well-formedness* of expressions. We use the term `type` to provide a well-formedness type judgment. To form this judgment we create a module `Fol_type`.

• *Create the file* `fol_type.mli` *containing the lines:*

```
extends Base_theory

declare "type"{'A}
```

The intent of the "`type`" term is that if the expression `type{'t}` is provable, then the term `'t` is a well-formed formula.

Next, we create the implementation. The rules for well-formedness will be added separately to the modules that define the logical operators, so the only implementation we need to include is the syntax declaration and a display form.

• *Create the file* `fol_type.ml`*, containing the following lines:*

```
extends Base_theory

declare "type"{'A}
declare trivial

dform type_df : "type"{'A} = slot{'A} `" type"
dform trivial_df : trivial = cdot
```

Now we will need to add well-formedness rules to the existing `fol` modules (you will also need to add `extends Fol_type` to each of them).

First, the logical constants are always well-formed.

• *Add a rule stating that* `true` `false` *are well-formed to* `Fol_true` *and* `Fol_false` *respectively:*

```
extends Fol_type

prim false_type {| intro [] |} :
   sequent { <H> >- "type"{."false"} } = trivial

extends Fol_type

prim true_type {| intro [] |} :
   sequent { <H> >- "type"{."true"} } = trivial
```

Next, a logical operation is well-formed whenever all the operands are well-formed.
• *Insert a rule describing the well-formedness of implication (in front of other rules) into* `Fol_implies`*:*

```
extends Fol_type

prim implies_type {| intro [] |} :
   [wf] sequent { <H> >- "type"{'A} } -->
   [wf] sequent { <H> >- "type"{'B} } -->
   sequent { <H> >- "type"{implies{'A; 'B}} }
   = trivial
```

For the sequents, the standard semantics is that for a sequent to be valid, its conclusion needs to be well-formed whenever all the hypotheses are well-formed. This means the existing rules will be valid with respect to well-formedness, unless one of the terms moves from the conclusion of the goal into a hypothesis in one of the assumptions. Such rules should be augmented with extra well-wormedness assumption(s).
• *Modify the* `implies_intro` *rule as follows*:

```
prim implies_intro {| intro [] |} :
   [wf] sequent { <H> >- "type"{'A} } -->
   ('b['x] : sequent { <H>; x: 'A >- 'B }) -->
   sequent { <H> >- 'A => 'B }
   = lambda{x. 'b['x]}
```

In this module we have also added *label* annotations to the rules with prefixes of the form `[wf]` and `[main]`. These labels are assigned to the subgoals during refinement, and are displayed as labels on the proofs.
• *In a similar fashion, modify the remaining modules to contain the following rules:*

```
prim and_type {| intro [] |}:
   [wf] sequent { <H> >- "type"{'A} } -->
   [wf] sequent { <H> >- "type"{'B} } -->
   sequent { <H> >- "type"{.'A & 'B} } = trivial

prim or_type {| intro [] |} :
   [wf] sequent { <H> >- "type"{'A} } -->
   [wf] sequent { <H> >- "type"{'B} } -->
   sequent { <H> >- "type"{."or"{'A; 'B}} } =
   trivial

prim or_intro_left
   {| intro [SelectOption 1] |} :
   [wf] sequent { <H> >- "type"{'B} } -->
   [main] ('a : sequent { <H> >- 'A }) -->
```

```
   sequent { <H> >- "or"{'A; 'B} } =
   inl{'a}

prim or_intro_right
   {| intro [SelectOption 2] |} :
   [wf] sequent { <H> >- "type"{'A} } -->
   [main] ('b : sequent { <H> >- 'B } ) -->
   sequent { <H> >- "or"{'A; 'B} } =
   inr{'b}

interactive not_type {| intro [] |} :
 [wf] sequent { <H> >- "type"{'A} } -->
 sequent { <H> >- "type"{."not"{'A}} }

interactive not_intro {| intro [] |} :
 [wf] sequent { <H> >- "type"{'A} } -->
 sequent { <H>; 'A >- "false" } -->
 sequent { <H> >- "not"{'A} }

prim pred_type {| elim [] |} 'H :
   sequent { <H>; x: pred; <J['x]> >- "type"{'x} }
   = trivial

prim all_type {| intro [] |} :
   [wf] sequent
      { <H>; x: pred >- "type"{'B['x]} } -->
   sequent
       { <H> >- "type"{."all"{y. 'B['y]}} }
       = trivial

prim all_intro {| intro [] |} 'H :
   [main] ('b['x] : sequent
      { <H>; x: pred >- 'B['x] }) -->
   [wf] sequent
      { <H>; x: pred >- "type"{'B['x]} } -->
   sequent { <H> >- "all"{y. 'B['y]} } =
   lambda{y. 'b['y]}

prim all_elim {| elim [ThinOption thinT] |} 'H 'a :
 [wf] sequent
  { <H>; x: "all"{y. 'B['y]}; <J['x]> >- "type"{'a} } -->
 [main] ('b['x; 'z] : sequent
  { <H>; x: "all"{y. 'B['y]};
    <J['x]>; z: 'B['a] >- 'C['x] }) -->
 sequent { <H>; x: "all"{y. 'B['y]};
    <J['x]> >- 'C['x] } =
 'b['x; 'x 'a]
```

Table 29: The interpretation of the `false` constant

```
extends Itt_theory
extends Fol_itt_type

derive Fol_false

prim_rw unfold_false : "false" <--> void

derived false_type :
   sequent { <H> >- "type"{."false"} }

derived false_elim 'H :
   sequent { <H>; x: "false"; <J['x]> >- 'C['x] }
```

4.2 Deriving FOL from type theory

We'll start by creating a `Fol_itt_type` module deriving `Fol_type` with the following code. The first step includes the `Itt_theory` module, which defines the Nuprl type theory. The next step, `derive Fol_type` states that the `Fol_type` module is to be *derived*, and that all its rules become proof obligations. The `Fol_type` module has no rules, so in this case there are no extra obligations. The final statement, a `prim_rw` definition, provides the definition of the type judgment in the `Fol_type` module directly in terms of the type judgment in the `Itt_equal` module in the Nuprl type theory.

```
extends Itt_theory
derive Fol_type
prim_rw unfold_type : Fol_type!"type"{'t} <-->
    Itt_equal!"type"{'t}
```

The logical operators are a little more interesting. To derive the `Fol_false` module, we need to give an interpretation of the logical constant `false`. In keeping with the Curry-Howard isomorphism, the constant `false` should be represented by an *empty* type (there are no proofs of `false`). The code for the `Fol_itt_false` module is shown in Table 29.

The `Fol_itt_type` module is included for the definition of the `type` judgment, and our goal is to derive the module `Fol_false`. The rewrite gives the definition of `false` in terms of the `void` type from the module `Itt_void` (in this case the unqualified names are unambiguous). The next step is to derive the rules from the `Fol_false` module.[1] To the editor, the `derived` statement for a rule is treated the same as if the rule were declared as `interactive`: it determines a proof obligation that must be satisfied with the editor. So the next question is: how do we prove the `false_type` and `false_elim` rules?

[1] At present, the derivations must be stated explicitly. This may go away in future version of the system, since the derivations may be inferred from the `derive Fol_false` statement.

Table 30: Proving the `false_type` rule

Step 1
| # [*]
| ....main....
| $< \Gamma >$ ⊢ false type
| **BY** rwh unfold_false 0 thenT rwh unfold_type 0
| **1. [#]**
| ....main....
| $< \Gamma >$ ⊢ Void Type

Step 2
| * * [*]
| ....main....
| $< \Gamma >$ ⊢ Void Type
| **BY** dT 0

Table 31: Constructive interpretation of the logical operators

Operator	Interpretation
`type{'t}`	`Itt_equal!type{'t}`
`false`	`Itt_void!void`
`true`	`Itt_unit!unit`
`and{'A; 'B}`	`Itt_prod!prod{'A; 'B}`
`or{'A; 'B}`	`Itt_union!union{'A; 'B}`
`implies{'A; 'B}`	`Itt_fun!fun{'A; 'B}`

In general, the first step in a derivation is to unfold the definitions of the defined terms—in this case `Fol_type!type` and `Fol_false!false`. The proof is shown in Table 30.

Step 1 replaces the definition of `false` with `void` and `Fol_type!type` (which displays the term `type{`t`}` as t `type`) with `Itt_equal!type` (which displays the term `type{`t`}` as t `Type`). The Nuprl type theory is automated in much the same way as we provided the automation for the first-order logic, and we can use the `dT 0` tactic to knock off the subgoal, as shown in Step 2.

The interpretation for all the logical operators is shown in Table 31. The constant `true` is interpreted as the type `unit`, which contains exactly one proof $\cdot$. For the propositional operators, `and` is interpreted as a Cartesian product, `or` as a disjoint union, and `implies` as a function space.

The modules for the propositional operators are all similar. The `Fol_and` module, listed in Table 32, shows the common outline. The first part establishes that this is a derivation from `Itt_theory` and `Fol_itt_type` modules to the module `Fol_and`.

Table 32: Listing of the `Fol_itt_and` module

```
extends Itt_theory
extends Fol_itt_type

derive Fol_and

open Tactic_type.Conversionals

prim_rw unfold_and : "and"{'A; 'B} <--> prod{'A; 'B}
prim_rw unfold_pair :
   Fol_and!pair{'a; 'b} <--> Itt_dprod!pair{'a; 'b}
prim_rw unfold_spread :
   Fol_and!spread{'a; x, y. 'b['x; 'y]} <-->
   Itt_dprod!spread{'a; x, y. 'b['x; 'y]}

let fold_and = makeFoldC << Fol_and!"and"{'A; 'B} >> unfold_and
let fold_pair =
   makeFoldC << Fol_and!"pair"{'a; 'b} >> unfold_pair
let fold_spread = makeFoldC
   << Fol_and!spread{'a; x, y. 'b['x; 'y]} >> unfold_spread

derived_rw reduce_spread :
   spread{pair{'x; 'y}; a, b. 'body['a; 'b]} <--> 'body['x; 'y]

derived and_type :
   [wf] sequent { <H> >- "type"{'A} } -->
   [wf] sequent { <H> >- "type"{'B} } -->
   sequent { <H> >- "type"{.Fol_and!"and"{'A; 'B}} }

derived and_intro :
   [main] ('a : sequent { <H> >- 'A }) -->
   [main] ('b : sequent { <H> >- 'B }) -->
   sequent { <H> >- Fol_and!"and"{'A; 'B} }

derived and_elim 'H :
   [main] ('body['y; 'z] :
      sequent { <H>; y: 'A; z: 'B; <J[Fol_and!pair{'y; 'z}]> >-
         'C[Fol_and!pair{'y; 'z}] }) -->
   sequent { <H>; x: Fol_and!"and"{'A; 'B}; <J['x]> >- 'C['x] }
```

Table 33: Prove the `and_type` theorem

Step 1
* [*]
....main....
1. $< \Gamma > \vdash A$ type
2. $< \Gamma > \vdash B$ type
====
$< \Gamma > \vdash (A \wedge B)$ type
BY rwh unfold_and 0 thenT rwh unfold_type 0 thenT dT 0
1. [*]
....wf....
$[\cdot] < \Gamma > \vdash A$ Type
2. [*]
....wf....
$[\cdot] < \Gamma > \vdash B$ Type

Step 2
* * [*]
....wf....
1. $< \Gamma > \vdash A$ type
2. $< \Gamma > \vdash B$ type
====
$[\cdot] < \Gamma > \vdash A$ Type
BY rwh fold_type 0 thenT unsquashT « 'ext » thenT trivialT

The rewrites provide the definitions of the operators: the conjunction is the Cartesian product, and the proof terms have the corresponding representations. Rewrite definitions define conversions that operate from left-to-right: the `unfold_and` conversion rewrites a conjunction to a product space. It is also useful to have the right-to-left version, called `fol_and`, that rewrites the product space to the conjunction. The `makeFoldC` conversional (defined in `Tactic_type.Conversionals` build a conversion for right-to-left rewriting given the redex term and the rewrite definition.

The last part of the `Fol_itt_and` module states the rules that are to be derived. To make an arbitrary choice, let's look at the proof of the theorem `and_type`. The initial step is to unfold the definitions for the conjunction and type judgment, shown in Table 33, Step 1. This refinement produces two similar subgoals, both of which are similar (but not identical to) the assumptions.

The proof of one of these subgoals requires *folding* the definition of the `type` judgment term. It also requires an `unsquashT` operation: the optional argument to the sequent in the subgoal is the term `squash` $(\cdot)$, rather than the conventional `'ext` variable, and the `unsquashT` tactic will perform the replacement. The proof for the first subgoal is shown in Step 2.

Table 34: Providing classical reasoning

```
extends Fol_not

declare magic{x. 't['x]}

dform magic_df : magic {x. 't} = `"magic"

prim magic :
   ('t['x] : sequent
      { <H>; x: "not"{'T} >- "false" }) -->
   sequent { <H> >- 'T } =
   magic{x. 't['x]}

let magicT = magic
```

Unfortunately, this interpretation breaks down for the quantifiers because the first-order logic is *impredicative* and the type theory is *predicative*. Briefly, this means that quantification in the type theory is strictly inductively defined, and there is no non-trivial type that includes itself as a member. In contrast, in the first-order logic, a predicate like $\forall P.P \Rightarrow P$ is quantified over *all* predicates including the predicate itself.

Instead, we will leave the derivation as it stands (for the propositional logic only), and we will skip to developing a *classical* first-order logic.

5 Defining classical first-order logic

Classical logic is a conservative extension to the constructive logic we have just defined (all the existing rules are still valid). In a classical setting, every predicate is either *true* or *false*. We can enable classical reasoning in this single-conclusion logic with a single rule, based on double-negation elimination $\neg\neg P \Rightarrow P$. The proof term for this proposition has no computational meaning, so we provide a new term `magic` to represent proofs by excluded middle.

The implementation `cfol_magic.ml` is shown in Table 34. The module `Fol_not` is required for the definition of negation. The `magic` rule essentially states the following: to prove a predicate T, assume that T is false and prove a contradiction. The `magicT` tactic provides the standard tactic wrapping functionality.

The classical theory can be pulled together as a module with just two lines that extend the constructive version of the logic together with the `magic` module.

```
extends Fol_theory
extends Cfol_magic
```

6 Deriving classical first-order logic from the Nuprl type theory

Deriving the classical logic from type theory is significantly different from deriving the constructive logic. The main obstacle is that the Nuprl type theory is constructive, and there is no way to justify the `magic` rule directly. Instead, we will build a *model* of classical logic in the type theory, rather than coding it directly using the most obvious type constructors.

Another obstacle is that in the classical system, every predicate is either *true* or it is *false*, and it is natural to consider two predicates to be equal when they are either both true or both false.

Initially, we can start along the same path as the interpretation for the constructive logic. First, we intend to use the types `void` and `unit` as the canonical representatives for `false` and `true`. However, if we make these choices, we will have problems interpreting the propositional connectives. For example, the obvious interpretation of $A \wedge B$ is the product space $A \times B$. But this breaks equality reasoning: $\texttt{true} \wedge \texttt{true} = \texttt{unit} \times \texttt{unit} \neq \texttt{true}$. In some sense, the problems arises because of the computational content of the types: a product space contains *pairs*, while the `unit` type contains only the canonical term $\cdot$.

To solve this problem, we use a standard Nuprl technique for *computational hiding*. The $\texttt{esquash}\{T\}$ term is a type for any type T, and it contains the unique canonical element $\cdot$ if T is inhabited, and it is empty otherwise. Furthermore, $\texttt{esquash}\{T_1\} = \texttt{esquash}\{T_2\}$ whenever $T_1 \Leftrightarrow T_2$. The usual display notation for $\texttt{esquash}\{T\}$ is $\downarrow T$.

The initial starting point for the interpretation is then as follows. The interpretation of `false` is the type $\downarrow \texttt{void}$ and the interpretation of `true` is $\downarrow \texttt{unit}$. The $\texttt{type}\{P\}$ judgment enforces the condition that the predicate P be either true or false with the interpretation $\downarrow (P = \mathit{true} \in \mathbb{U}_1 \vee P = \mathit{false} \in \mathbb{U}_1)$.

We will define this initial part of the interpretation in the module `Cfol_itt_base`. The module definition is shown in two parts, in tables 35 and 36. The derivation is for the four modules `Fol_type`, `Fol_false`, `Fol_true`, and `Fol_pred` all at once. The definitions for false and true are the squashed versions of void and unit, and the type judgment is the squashed disjunction stating that the argument predicate is either false or true. The `pred` type is the set of types that are predicates. The next few theorems are convenient lemmas stating some relationships between the predicates in the types in `univ[1:l]`.

The next step is to derives the rules for `false`, `true`, and `pred`. These proofs are fairly straightforward. Note that in this interpretation we get the additional facts that `true` and `false` are members of the `pred` type.

The interpretation of the logical operators is shown in Table 38. The proofs are fairly long. The `all_elim` proof illustrates the kinds of reasoning involved; Table 39 shows the initial goal in the proof. The basic strategy is as follows: first, $\forall v.B[v]$ is equivalent to $B[\mathit{false}] \wedge B[\mathit{true}]$. Since a is either true or false, then $B[a]$ is equivalent to either $B[\mathit{false}]$ or $B[\mathit{true}]$, so $B[a]$ follows by assumption, and we can add it to the sequent using the `cut` rule.

Table 35: Base interpretation `Cfol_itt_base` (Part 1)

```
extends Itt_theory

derive Fol_type
derive Fol_false
derive Fol_true
derive Fol_pred

open Tactic_type
open Tactic_type.Conversionals
open Tactic_type.Tacticals

open Dtactic

open Itt_equal

(* Interpretation *)
prim_rw unfold_false :
   Fol_false!"false" <--> esquash{void}
prim_rw unfold_true :
   Fol_true!"true" <--> esquash{unit}
prim_rw unfold_type : Fol_type!"type"{'t} <-->
   squash{(('t = "false" in univ[1:l]) or
      ('t = "true" in univ[1:l]))}
prim_rw unfold_pred : Fol_pred!"pred" <-->
   { T: univ[1:l] | "type"{'T} }

let fold_false =
    makeFoldC << Fol_false!"false" >> unfold_false
let fold_true =
    makeFoldC << Fol_true!"true" >> unfold_true
let fold_type =
    makeFoldC << Fol_type!"type"{'t} >> unfold_type
let fold_pred =
    makeFoldC << Fol_pred!"pred" >> unfold_pred

(* Lemmas *)
interactive false_univ {| intro [] |} :
   sequent { <H> >- "false" IN univ[1:l] }

interactive true_univ {| intro [] |} :
   sequent { <H> >- "true" IN univ[1:l] }

interactive type_univ {| intro [AutoMustComplete] |} :
   [wf] sequent { <H> >- 'A Type } -->
   sequent { <H> >- 'A IN univ[1:l] }

let typeT = type_univ
```

Table 36: Base interpretation `Cfol_itt_base` (Part 2)

```
(* Rules for false *)
derived false_type {| intro [] |} :
   sequent { <H> >- "false" Type }

derived false_elim {| elim [] |} 'H :
   sequent { <H>; x: "false"; <J['x]> >- 'C['x] }

(* Rules for true *)
derived true_type {| intro [] |} :
   sequent { <H> >- "type"{."true"} }

derived true_intro {| intro [] |} :
   sequent { <H> >- "true" }

interactive true_concl_elim :
   [main] sequent
      { <H> >- 'P = "true" in univ[1:l] } -->
   sequent { <H> >- 'P }

interactive true_concl_intro :
   [wf] sequent { <H> >- 'P Type } -->
   [main] sequent { <H> >- 'P } -->
   sequent { <H> >- 'P = "true" in univ[1:l] }

let trueT = funT (fun p ->
   let goal = Sequent.concl p in
      if is_equal_term goal then
         true_concl_intro
      else
         true_concl_elim)
```

Table 37: Base interpretation `Cfol_itt_base` (Part 3)

```
(* Rules for pred *)
interactive pred_elim {| elim [] |} 'H :
   sequent { <H>; x: univ[1:l]; y: "type"{'x};
      <J['x]> >- 'C['x] } -->
   sequent { <H>; x: pred; <J['x]> >- 'C['x] }

interactive true_pred {| intro [] |} :
   sequent { <H> >- "true" IN pred }

interactive false_pred {| intro [] |} :
   sequent { <H> >- "false" IN pred }

interactive pred_type1 {| intro [] |} :
   sequent { <H> >- Itt_equal!"type"{pred} }

derived pred_type {| elim [] |} 'H :
   sequent { <H>; x: pred; <J['x]> >- "type"{'x} }
```

Table 38: Interpretation of the classical operators

`false`	`esquash{void}`
`true`	`esquash{unit}`
`type{'t}`	`esquash{ union{'t = false in univ[1:l]; 't = true in univ[1:l]}}`
`pred`	`{ T: univ[1:l] | type{'T} }`
`and{'A; 'B}`	`esquash{prod{'A; 'B}}`
`or{'A; 'B}`	`esquash{union{'A; 'B}}`
`implies{'A; 'B}`	`esquash{fun{'A; 'B}}`
`all{v. 'B['v]}`	`and{'B[false]; 'B[true]}`
`exists{v. 'B['v]}`	`or{'B[false]; 'B[true]}`

Table 39: Prove the `all_elim` theorem

Step 1
! [**!**]
....main....
1. $< \Gamma >$; x: $\forall y.\ B[y]$; $< \Delta[x] >\vdash\ a$ type
2. $< \Gamma >$; x: $\forall y.\ B[y]$; $< \Delta[x] >$; z: Pred$\vdash\ B[z]$ type
3. $< \Gamma >$; x: $\forall y.\ B[y]$; $< \Delta[x] >$; z: $B[a]\vdash\ C[x]$
====
$< \Gamma >$; x: $\forall y.\ B[y]$; $< \Delta[x] >\vdash\ C[x]$
BY assertT « 'B['a] » thenMT autoT
1. [!]
....main....
$< \Gamma >$; x: $\forall y.\ B[y]$; $< \Delta[x] >\vdash\ B[a]$

7 Conclusion

This concludes definition of the first-order logic. We have seen:

- how to define a primitive logic, extend constructive and classical versions of first-order logic,
- how to add automation and decisions procedures to a logic,
- how to use resources to simplify the task of adding proof automation,
- how to define a theory *relation*—in particular how to derive the first-order logics from the Nuprl type theory.

First-order logic is a fairly simple logic; the definition of the logic and its automation is fairly straightforward. We are hoping that it served as a good illustration and introduction on MetaPRL features and and got you ready to try implementing something more exciting and challenging.