Using Twelf to Prove Type Theorems

John Tang Boyland

February 27, 2007

Abstract

Twelf is an implementation of the LF logical framework proposed as a vehicle for writing machine-checked proofs of type system theorems. This paper serves several purposes: it records experience from an outsider trying to learn the system; it contains an introduction to using Twelf; it can serve as a tutorial for learning Twelf.

1 Introduction

The "POPLmark" challenge is looking toward a time when papers presented at conferences such as POPL are routinely accompanied by machine-checked proofs. Having written several type-theory papers with extensive hand-written proofs, the idea of having machine-checked proofs is attractive. In particular, the ability to reuse proofs after minor variations is desirable. However learning to use a proof system is daunting. In this paper, I give my experience in starting to learn how to use Twelf (specifically version 1.5R1). At the time of writing, I had been using Twelf for little more than two weeks. There is a general paucity of information directed to beginning Twelf users interested in proving the soundness of type systems. Thus despite my meager knowledge, I felt that I should share my thoughts with others. I will do my best to correct any errors and misleading information in this text pointed out by Twelf masters.

1.1 Why Twelf?

Part of the POPLmark effort consists of posting solutions to the challenge problems. The Twelf solution was not only one of the submissions that came closest to meeting the challenge (remarkably, no solution fully addresses all parts of the challenge) but also was extensively commented.

To a neophyte such as myself, the main contending systems appear to be Isabelle/HOL and Twelf. Of the two, Isabelle/HOL is by far the better documented and user friendly. However, the ability to express proofs in formal language made Twelf much more attractive, whereas from the Isabelle documentation, it appeared a complete proof in Isabelle consists of a list of proof tactices to apply—not the proof itself. It seems that in a theorem proving system, there is less interest in the proof itself than in the fact that it is proved. From other information (in particular the slides by Christian Urban), it appears possible to express proofs in Isabelle's meta-syntax ISAR, but this aspect of Isabelle is, if possible, documented even worse than Twelf. The proofs produced by ISAR look more like natural language proofs, with all the benefits and problems that leads to. On the other hand, Twelf proofs look like Prolog programs. Perhaps the programmer in me is attracted to the latter, but in any case I first started investigating Twelf.

1.2 Twelf's Documentation

Twelf comes wih a "user guide" which includes more information than one is likely to find elsewhere. At the time I write this, the user guide has not been updated for Twelf 1.5 and so I used the user guide for Twelf 1.4.

The user guide acknoweldges that both it and the Twelf implementation are research products. The user guide was apparently written to serve as an introduction as well as reference manual, but shows signs of decay: the formal reference elements are actively maintained but the explanatory material is changed only when it is actively contradicted. Thus it would have one believe that theorem proving (as opposed to theorem checking) is a recent experimental addition to Twelf when it dates at least to Twelf 1.2 (1998), and apparently predates the theorem checker (first in Twelf 1.4) (2002).

The Twelf web page `http://www.cs.cmu.edu/~twelf` also contains links to Pfenning's book *Computation and Deduction* and a tutorial by Appel. The tutorial show how to use the Twelf system to prove theorems in a logic embedded in Twelf, rather than showing how to use Twelf to prove theorems directly. The book has some valuable information and exercises in it but includes little on using either the theorem checker or theorem prover. There is also a Twelf Wiki `http://fp.logosphere.cs.cmu.edu/twelf`, at which questions can be posted and often get answered quickly. My thanks to all those who have provided introductory material for Twelf. My dependence on them and on Pierce's *Type Systems for Programming Languages* will be visible in my examples.

2 Twelf Basics

If one wishes to use Twelf to prove type safety theorems, one will need to write the following aspects:

- The abstract syntax of terms and type. This aspect comprises two kinds of declarations: *nonterminals* (such as "term") and *productions* (such as "application := term term").
- The evaluation and type relations. Again, this aspect comprises two kinds of declarations: *signatures* of a relation (e.g., $\vdash t : \tau$) and *rules* (e.g., UNIT $\vdash$ `()` : unit);
- The lemmas and theorems about the relations. Yet again, we have two kinds of declarations: the statement of the theorem, and the proof (one "declaration" for each case).

Twelf elegantly (but confusingly) uses the same syntax to define both parts of all three aspects (six logically separate entities):

name : *type* .

The Twelf theorem prover does support a special syntax for expressing the statement of theorems, and from experiments, it appears this syntax can also be used for manually constructed proofs. On the other hand, I have been informed that the syntax is subject to modification at any time and its use should not be encouraged.

Twelf as an executable system is most conveniently invoked from within Emacs, although it can also be executed from within SML. I was initialized confused because the "twelf server" doesn't actually accept Twelf syntax. Rather Twelf syntax must be read indirectly, for example, using Emacs. The "twelf server" does has the ability to run "queries" interactively, as explained below.

To use this document as a tutorial, install Twelf according to directions (including installing the Emacs mode). Then create a fresh directory for sample files and create an empty file named `source.cfg`. Save it and then visit a file "test.elf" in which to write Twelf code. Start the server using Meta-X `twelf-server-configure`. Consent to any questions Emacs asks about configuring the Twelf server.

2.1 Abstract Syntax

A new abstract nonterminal N is declared by writing "N : `type.`" where `type` is the special type for types. New productions for this nonterminal are declared as functions that return this nonterminal. Thus, a simple expression language can be declared as follows:

```
term : type.

true : term.
```

```
false : term.
if : term -> term -> term -> term.
zero : term.
```

The `zero` term is included to ensure that the toy language has non-trivial typing. In Twelf, constants (everything declared so far) should not start with capital letters but may include a variety of symbols other than colons, periods or any kind of parenthesis. It would be legal to use `0` (ASCII zero) instead of `zero`.

To continue the tutorial, type in this "grammar" from memory in the Emacs buffer and then press Control-C Control-S. If you make a mistake in the syntax (such as forgetting a period), you will get an error and can use Control-C backquote to have the error highlighted. Correct the error and press Control-C Control-S again, until no errors are noted.

We also need to define the abstract syntax of types. The type system is very simple and thus can be declared as follows:

```
ty : type.

bool : ty.
int : ty.
```

Of course we can't use `type` as the type of our mini-language's types!

On a side-note: Twelf doesn't include polymorphism, which can be annoying at times: every new kind of list type must be redeclared.

For the tutorial, add the declarations of the types to your Twelf file from memory.

As well as declarations, Twelf also permits one to write *definitions*. Definitions at the syntactic level function as syntactic sugar. For example, in order to add syntactic sugar for "and," one can write:

```
%abbrev and : term -> term -> term = [B1] [B2] if B1 B2 false.
```

The square-bracketed variables are lambda bound variables. Variables should start with capital letters (although there are ways to avoid this).

The `%abbrev` tag is a strange requirement that I haven't fully understood. Basically it means that the name "and" will be immediately replaced by its definition wherever it is used. It is needed if the uses of the variables are not "strict," according to a technical definition of "strictness." In this case, the uses are "strict." I find however that even if all uses are strict, this tag is still obligatory to avoid cryptic errors in the proof checker.

Exercise 1 Write a definition for `or`. □

The term following the equal sign (`=`) looks a little bit like a functional language expression. However, without conditionals or recursion, possibilities are limited. In particular operations on terms (such as “size” or sets of free variables) must be defined using relations instead. There are very good theoretical reasons for this restriction which I will not attempt to explain.

2.2 Defining relations

The next step in defining a type system is to define the relations (judgments) that one will prove properties about. First one defines the type of the relation itself as a function that produces a “type.” For example, suppose we wish to define a small-step evaluation relation $t \longrightarrow t'$. The arrow in the relation here is merely a convention, the relation could easily have be written $\langle t, t' \rangle$. In any case, the relation consists of two terms and thus we write:

```
eval : term -> term -> type.
```

Typically a relation has several different inference rules (axioms). These may be written in a style reminiscent of Prolog. For example, evaluating an “if” expression has the following rule:

E-If

$$\frac{e \longrightarrow e'}{\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \longrightarrow \texttt{if}\ e'\ \texttt{then}\ e_1\ \texttt{else}\ e_2}$$

In Twelf (using abstract syntax), we write this as follows

```
eval/if : eval (if E E1 E2) (if E’ E1 E2)
    <- eval E E’.
```

The inference rule must be given a unique name so that we can refer to it in proofs. Here the name is “`eval/if`” following a Twelf convention, separating teh relation name from the case name by a slash. The consequent of the judgment is given first and then the antedecents, each preceded by `<-`. In this case, the only antecedent is a “recursive call” to `eval`. Twelf is also perfectly happy to have the inference rules written the other way around, using `->`:

```
eval/if : eval E E’ -> eval (if E E1 E2) (if E’ E1 E2)
```

The original “backward” style is often preferred because it reflects Twelf’s operational semantics: order of evaluation. The forward style however makes proofs more natural since it reflects the order the antecedents are passed to the inference rule.

For the next step in the tutorial, write the declaration of “eval” and its first inference rule by memory and get Twelf to accept the whole file with Control-C Control-S.

Twelf is happy to accept `E'` as a variable name. It has no conception that there is any connection between this variable and the variable `E`. However, for the human reader, meaningful variable names help comprehension. Variables have types. In most cases, Twelf can infer the types automatically, but if not, you can write `(E:term)` to force `E` to have type `term`. In the `*twelf-server*` window, you can see what type it assigns:

```
eval/if : {E:term} {E':term} {E1:term} {E2:term}
      eval E E' -> eval (if E E1 E2) (if E' E1 E2).
```

Notice that the variable's type is presented in an abstraction introduced with curly braces (rather than square brackets). Technically this is because `eval/if` is given a "dependent type." For practical purposes, you can just think of it as a way to explicitly indicate the type of a variable. Indeed you can use this syntax to define this inference rule which will then have *six* parameters rather than just two when you use it in proofs. In the way we originally typed it, the four variables are *implicit parameters* to the inference rule. We will return to this topic later.

Next we define the evaluation rules for 'if" where the condition is fully evaluated:

```
eval/if_true : eval (if true X _) X.
eval/if_false : eval (if false _ X) X.
```

These rules have no antecedents. The underscore (`_`) is an anonymous variable in the style of Prolog.

One can test the evaluation rules using Twelf's query language. In Emacs, this can be done by visiting the `*twelf-server*` buffer and typing `top` at the end. This gets one a Prolog style "`?-`" prompt at which one can type a query (followed by period) such as

```
?- eval (if true false true) X.
Solving...
X = false.
More?
```

One can type `y` to see what other possibilities there are, or just press return. Here there are no other possibilities. If in tutorial mode, try to see what expressions could evaluate to `false`. Twelf will give two independent possibilities. To leave the interactive query system in Emacs, press Control-C TAB.

Exercise 2 Add a new production `iszero : term -> term` and evaluation rules for it that reflect the following inference rules:

E-IsZero

$$\frac{e \longrightarrow e'}{\texttt{iszero}\ e \longrightarrow \texttt{iszero}\ e'}$$

E-IsZeroZero

$$\frac{}{\texttt{iszero 0} \longrightarrow \texttt{true}}$$

Check that it works using Twelf's query system. □

Next, we define the typing relation: $\vdash e : \tau$. Again the elements $\vdash$ and : are conventional, what we really have is a relation with two parts, a term and a (mini-language) type:

```
of : term -> ty -> type.
```

The only non-trivial type rule is the one for `if`'s:

```
of/true : of true bool.
of/false : of false bool.
of/zero : of zero int.
of/if : of X bool -> of Y T -> of Z T -> of (if X Y Z) T.
```

Here I use the "forwards" style. Notice that the repeated use of the variable "T" ensures that each side of the "if" has the same type.

In tutorial mode, write these rules from memory and check that `of/if` has four implicit parameters.

Exercise 3 The inference rule for `iszero` is as follows:

$$\textsc{T-IsZero}\;\frac{\vdash e : \mathrm{int}}{\vdash \texttt{iszero}\ e : \mathrm{bool}}$$

Implement this inference rule in Twelf. □

2.3 Theorems

In type soundness proofs based on small-step semantics, *progress* means that any well-typed expression (with no free variables, but we don't have these here) is either already a value or else can be evaluated one step further, where *preservation* means that if any well-typed expression can be evaluated, the resulting expression is also well-typed with the same type. In order to express progress, therefore, we need to have some notion of a "value," a subset of the terms.

Since Twelf does not having subtyping, "valueness" must be defined as a relation:

```
is_value : term -> type.

is_value/true : is_value true.
is_value/false : is_value false.
is_value/zero : is_value zero.
```

Next, since the progress theorem says that the term is either a value or can be evaluated a further step, this “or” will need to be converted into an explicit relation that is true in either case:

```
not_stuck : term -> type.

not_stuck/value : not_stuck X <- is_value X.
not_stuck/eval : not_stuck X <- eval X X'.
```

It has two cases, no matter how many different evaluation rules or values we have.

Twelf accepts a special syntax for theorems (or lemmas, but they are all called theorems in Twelf). Here is how we can express the statement of the progress theorem:

```
%theorem
progress : forall* {X} {T}
           forall {O:of X T}
           exists {NS:not_stuck X}
           true.
```

Various parts can be omitted, but not repeated and not put in a different order. The keyword `forall*` has an asterisk because the variables thus mentioned are implicit—when using the theorem, one does not pass the corresponding terms as parameters. On the other hand, the next variable `O` is explicit with a relation type. The result is an explicit *output* parameter (result of the theorem) rather than *input* parameters (requirements of the theorem). Basically the theorem can be seen as a function that takes the input parameter (the typing fact) and returns the output parameter (the fact of non-stuck-ness). The theory behind Twelf is that if such a function is

well-typed according to Twelf's type system (LF), and is

well-moded in that it correctly “calls” any theorems it uses and defines its outputs, and is

terminating on all inputs, and is

total in that every input is mapped to some result,

then one has a valid theorem. A `%theorem` declaration succeeds as long as all variables are declared: a theorem is not allowed implicit free variables, implicit variables must be bound by `forall*`. Success says nothing as to whether the theorem is true or not!

To prove the theorem, one must either invoke the automatic theorem prover or else provide a proof. First, the theorem prover can be invoked using a `%prove` tag:

```
%prove 2 X-T (progress X-T _).
```

The number "2" is an indication of how hard you wish the theorem prover to try. Generally very simple inductive theorems can be proved with level 2, more complex ones with 3. I haven't found it necessary to use larger numbers than 3.

In the `%prove` request, the `X-T` is the variable on which induction should be tried. Its meaning is determined by the "call pattern" `(progress X-T _)` in which the other parameter is elided with an underscore. The natural language equivalent to this request is "Prove the progress theorem by structural induction over the typing derivation."

In tutorial mode, type in the theorem by memory and ask the theorem prover to prove it. In this case, (at least in Twelf 1.5R1), the theorem prover apparently diverges. After losing patience, press Control-C TAB to stop it. We will return to this theorem later after defining a helper lemma. In the mean time, comment out the `%prove` request by preceding it with `%%` (or by putting a space before the "`p`", an interesting convention I have seen people use).

The `%theorem` declaration is essentially equivalent to the following declarations (one can see these in the `*twelf-server*` window):

```
progress : {X:term} {T:ty} of X T -> not_stuck X -> type.
%mode +{X:term} +{T:ty} +{O:of X T} -{NS:not_stuck X} (progress O NS).
```

The first part is a normal relation declaration. The `%mode` declaration indicates which variables are input "+" and which are output "-". The inputs are universally quantified ("for all") and the outputs are existentially qualified ("there exists"). The mode declaration will cause further checks to occur when the proof is written.

The preservation theorem does not require a helper relation (as progress required "`not_stuck`"):

```
%theorem
preservation :
        forall* {X} {X'} {T}
        forall {O:of X T} {E:eval X X'}
        exists {O':of X' T}
        true.
```

If in tutorial mode, type in this theorem and ask the theorem to prove it, again by induction on the typing derivation. In this case, the call pattern has three explicit parameters. It should succeed quickly.

As we have seen, the theorem prover is not always capable of inferring a proof. In that case, one can either prove the theorem by hand (see next section) or else "assert" the truth of the theorem. Assertions can only be done once you have set "unsafe" mode (type `set unsafe true` at the end of the `*twelf-server*` buffer). Assertions are written in the following way:

```
%assert (progress _ _).
```

An asserted theorem can be used to (automatically) prove later theorems.

2.4 Writing proofs

An interesting start point for seeing how to write proofs would be to see the ones produced by the theorem checker. In Twelf 1.2, the theorem prover prints out the proof once it is found. In an unfortunate twist, this highly useful feature is *disabled* in later versions for technical reasons relating to higher-order syntax. I have been assured that this feature will be re-implemented soon; it is an embarrassment that the theorem prover in a system intended as a foundation for proof-carrying code won't produce the proofs it creates.

In any case, we still need to discuss the format for a proof. As explained above, a theorem is a relation that is indeed a function from inputs to outputs. The theorem cases are thus very much like the clauses implementing the inference rules, except that we are not interested in the name of the case. Starting with the preservation theorem, we can thus handle the case where evaluation has performed `eval/if_true`:

```
- : preservation (of/if (of/true) Y-T Z-T) (eval/if_true) Y-T.
```

The single hyphen ("`-`") is the uninteresting name of the clause (an alternate name might be `-eval/if_true`, using a prefix hyphen to ensure we don't shadow the previously defined relation case).

After the colon, we have the name of the theorem being proved and its "parameters." The first parameter is the typing relation. Since we know that the term is an "if" (because the evaluation rule is `eval/if_true`), the typing relation must use `of/if`. Now, it would make intuitive sense if the typing relation included the type and the term, but in Twelf, the relation doesn't include the pieces (its *type* does!), instead you must state *how* one knows the relation is true, in this case the antecedents of the T-IF type rule. The first antecedent is the typing relation for the condition (the literal term "`true`" because we are doing the case for `eval/if_true`). This relation must have been produced by `of/true`, the only rule to assign a type to `true`.

The variables `Y-T` and `Z-T` stand for the other two typing relations (*not* the subterms being typed!). Then the proof case states the evaluation rule, which is `eval/if_true` as explained before. It takes no parameters since the evaluation rule is unconditional, I have placed it (and the earlier unconditional rule `of/true`) in parentheses to emphasize their lack of parameters, but the parentheses are extraneous and may be omitted. The final parameter to `preservation` is the output parameter, the typing of the result. Since the result is the second child of the "if" term and its typing is the antecendent already bound as `Y-T`, we simply name the variable again. Of course the name `Y-T` is merely conventional. I would have chosen `Y:T` except that colons (unlike hyphens) may not appear in identifiers.

The case of `eval/if_false` is analogous:

```
- : preservation (of/if of/false Y-T Z-T) eval/if_false Z-T.
```

The case for E-If is more complex:

```
- : preservation (of/if X-bool Y-T Z-T) (eval/if X->X')
                 (of/if X'-bool Y-T Z-T)
     <- preservation X-bool X->X' X'-bool.
```

Unlike the previous cases, `eval/if` has an antecedent, which I have given the suggestive name `X->X'`. If we closed the rule before the `<-`, we would have a case that had a valid type, but would nonetheless give an error:

```
Occurrence of variable X'-bool in output (-) argument
not necessarily ground
```

If in tutorial mode, try this out by replacing the line starting `<-` with a single period.

The error comes from mode checking. The first two parameters to the theorem `preservation` are input parameters and thus `X-Bool` and `X->X'` get values from the inputs, through pattern matching. But the last parameter requires us to "call" `of/if`. Of its three parameters, the second is not yet determined, hence the error. The required typing relation is obtained using a recursive call to the theorem, that is, an *inductive* use of the theorem being proved.

In general, one may need to call other theorems as well (and thus have multiple clauses preceded by `<-`). Order is relevant because of mode checking. The backward syntax (that using `<-`) is preferred because then evaluation order coincides with textual order. You will notice however, that Twelf uses the forward syntax when echoing the cases. When trying to trouble-shoot errors, you will need to remember to mentally reverse the order when reading the generated output.

Exercise 4 Write the preservation cases for `iszero` evaluation. □

Once you think you've implemented all the cases, you can check that your theorem is "total" (that it covers all cases, and terminates successfully on all inputs):

```
%worlds () (preservation X-T X->X' %{=>}% X'-T).
%total X-T (preservation X-T _ _).
```

The `%worlds` declaration is required for reasons related to "higher-order abstract syntax." The "`()`" is a "trivial" world. In my opinion, the trivial worlds declaration should have been the default. However, since there is no default, I make use of the requirement to state a call pattern by giving suggestive names to the parameters, separating the input parameters from the output parameters by a comment (`%{=>}%`). I have the habit of writing the `%worlds` declaration before writing the cases; it helps serve as a reminder of the purpose of the parameters. On the other hand, this habit may run into "auto-freezing" problems which you may notice in a version other than Twelf 1.5R1.

The `%total` command requests that the theorem be checked using all the cases that Twelf is aware of. If you use the Emacs command Control-C Control-D to execute single Twelf lines at a time, then you must be careful to redeclare the theorem before checking the proof again or else any bad cases will still be present.

If in tutorial mode, check that the preservation theorem is total. Fix any mistakes in stating the cases.

The call-pattern is used in the same way as for the theorem solver—the `X-T` means that the theorem is proved inductively on the first "parameter." Twelf only uses structural induction. If you want to write a proof inductively on the size of something, you will need to reify the size as a natural number and then do structural induction on the natural number. See the next section for more information on doing this.

Often the term "structural induction" is used only for structural induction on terms, but as we have seen, relations are basically indistinguishable from terms in Twelf. Again, it's important to stress that the "children" of a relation are *not* the related terms, but rather the antecedents in the inference rule that proves the relation.

Twelf supports various kinds of mutual induction among parameters. See the "user guide" for a listing of different kinds of "termination orders."

Moving ahead to the "progress" theorem, we write the "worlds" declaration and the trivial value cases:

```
%worlds () (progress X-T %{=>}% X->X').
- : progress (of/true) (not_stuck/value is_value/true).
- : progress (of/false) (not_stuck/value is_value/false).
- : progress (of/zero) (not_stuck/value is_value/zero).
```

If in tutorial mode, type in a worlds declaration (no need to get fancy with the parameters) and these three cases from memory and give them to the Twelf server. Fix any errors.

The case for `if` must be handled differently. The problem is that we need to recursively check progress on the boolean condition and then depending on the result of this progress (the condition may be a value or may evaluate further) we need to do different actions. Unlike a hand-written proof, Twelf does not permit cases analysis inside another case analysis. Thus we need to perform what is called "output factoring" and declare a helper theorem on the output of our recursive "call":

```
%theorem
progress-if :
        forall* {X} {Y} {Z}
        forall {O:of X bool} {NS:not_stuck X}
        exists {NS':not_stuck (if X Y Z)}
        true.
```

Then we declare the trivial worlds for this theorem and give the cases:

```
%worlds () (progress-if X-T X->X' %{=>}% IFX->_).
- : progress-if _ (not_stuck/value is_value/true)
                  (not_stuck/eval eval/if_true).
- : progress-if _ (not_stuck/value is_value/false)
                  (not_stuck/eval eval/if_false).
- : progress-if _ (not_stuck/eval X->X')
                  (not_stuck/eval (eval/if X->X')).
```

In no case do we actually use the typing fact. The typing is needed only to avoid the need for a "`zero`" case. Twelf's coverage checker will be able to figure out that there's no way to give `zero` the type `bool`.

If in tutorial mode, type in these cases from memory, and then write a `%total` directive and check we have indeed proved the helper theorem.

Exercise 5 Write the progress proof cases for `iszero`. You will need to do "output factoring" again (write a helper lemma). □

We can now prove the theorem's totality:

```
%total X-T (progress X-T _).
```

If in tutorial mode, check that the progress theorem is correct.

The proof checker and theorem prover are basically independent modules in Twelf and neither has knowledge of the other. In other words, the theorem prover is unable to use a theorem proved manually and a manual proof is unable to use theorems proved automatically. This is another embarrassment, only partially mitigated by a new "unsafe" tag `%trustme` (not in Twelf 1.5R1) for asserting theorems to the totality checker. I believe that rectifying this situation is also high priority to the Twelf developers.

3 Numbers, Equality and Reduction

The Twelf user guide mentions "constraint domains" and in particular extensions to support integers and rational numbers directly. This seems very attractive, but fails in a bad way if you are trying to prove theorems: at best one gets cryptic "definition violation" error messages, at worst unsoundness results. Do not use "constraint domains" in conjunction with proofs.

Since Twelf does not currently come with a proof-friendly definition of numbers, one needs to "roll one's own." I intend to make available a "signature" of useful theorems concerning natural numbers. The examples here come from that signature. This section no longer has explicit tutorial directions, but the reader is recommended to try out the examples.

3.1 Introduction to Natural Numbers

Natural numbers are defined with a zero term and a successor term:

```
nat : type.
z : nat.
s : nat -> nat.
```

We then can define addition and subtraction (the latter an abbreviation):

```
plus : nat -> nat -> nat -> type.
plus/z : plus z Y Y.
plus/s : plus (s X) Y (s Z)
    <- plus X Y Z.

%abbrev
minus : nat -> nat -> nat -> type
      = [X] [Y] [Z] plus Z Y X.
```

Exercise 6 Define `times` and `divide` in a similar way. The case `times_s` will need two antecedents. □

Then we can prove that addition is associative:

```
%theorem
plus-associative :
        forall* {X:nat} {Y:nat} {Z:nat} {X':nat} {Z':nat}
        forall {P1:plus X Y X'} {P2:plus X' Z Z'}
        exists {Y':nat} {Q1:plus Y Z Y'} {Q2:plus X Y' Z'}
        true.
%prove 2 P1 (plus-associative P1 _ _ _ _).
```

I use hyphens to separate words in theorem names to distinguish them from relation cases (separated by slashes) and from multiple word relations (separated by underscores).

Exercise 7 Write out a proof for associativity that the Twelf theorem checker will accept. You will need two cases: one for the case that we `X` is zero (the addition fact uses `plus/z`), where we don't care what form `Y` (= `X'`) or `Z` has; and one for the case that `X` is not zero, and so the first addition fact must have been produced by `plus/s`, and thus the second addition fact must also have been produced by `plus/s`. The second case will use recursion (induction). □

Associativity, as it happens, is much easier to prove than commutativity:

Exercise 8 Write a statement of commutativity of addition and check that the theorem prover is *unable* to prove it. (One needs two inductive lemmas.) □

Indeed, as things get more complex, the theorem prover gets less useful. I found it was capable to proving most of the facts about addition but failed when attempting to prove facts about multiplication, even though the hand-written proofs were not that much longer. My guess is that the search space simply gets too big.

3.2 Equality

The associativity theorem presented above yields the addition fact `Y+Z=Y'` as an output parameter. Sometimes, however, one already knows this fact. Doing the "obvious thing" will get an "output coverage error" when checking the theorem using the associativity theorem in this way. I will explain this problem with a new theorem that assumes the addition fact in question and uses the original associativity theorem as a lemma:

```
%theorem
plus-associative* :
        forall* {X} {Y} {Z} {X'} {Y'} {Z'}
        forall {P1:plus X Y X'} {P2:plus X' Z Z'}
               {Q1:plus Y Z Y'}
        exists {Q2:plus X Y' Z'}
        true.
%worlds () (plus-associative* X+Y=X' X'+Z=Z' Y+Z=Y'
                          %{=>}% X+Y'=Z').
- : plus-associative*  X+Y=X' X'+Z=Z' Y+Z=Y' X+Y'=Z'
    <- plus-associative  X+Y=X' X'+Z=Z' Y' Y+Z=Y' X+Y'=Z'.
%total {} (plus-associative* _ _ _ _).
```

The induction argument `{}` tells the proof checker there will be no induction. This "proof" passes the type checker and the input coverage checker but fails totality with the following error message:

```
Occurrence of variable Y' in output (-) argument not free
```

The problem is not the occurrence of `Y'` by itself, but the implied occurrence in `Y+Z=Y'`. This addition fact is produced by `plus-associative`, but we already have an addition fact. The Twelf system has no way of knowing that the facts are guaranteed to have the same output; the output of a relation is not necessarily determined by the inputs. Essentially, we know `Y+Z=Y'` and the associativity theorem tells us `Y+Z=Y''` for a new variable `Y''`. We could ignore the fact using the new variable, but then the final result of the theorem (`X+Y''=Z'`) would unusable.

One solution is to prove the theorem over again from scratch, which is no great trouble since the theorem is easy to prove. On the other hand, these situations re-occur frequently in my experience. We need a way to show that `Y'` and `Y''` are the same. Thus we need a new relation defining equality:

```
eq : nat -> nat -> type.
eq/ : eq N N.
```

We don't need to worry about input vs. output coverage here because the *relation* is not moded, only the *theorems* about the relations. (It is possible to mode relations too, but I do not recommend that).

Then we need a theorem that addition is deterministic: it can only produce one possibility. (Twelf gives a way to force a moded relation to be deterministic, but the determinism is only available at the level of relations, not at the level of theorems about relations and thus it is useless for proofs.)

The theorem here is made more general as it accepts "equal" addends as well as proving an equal result:

```
%theorem
plus-deterministic:
        forall* {N1} {N1'} {N2} {N2'} {N3} {N3'}
        forall {P:plus N1 N2 N3} {P':plus N1' N2' N3'}
               {E1:eq N1 N1'} {E2:eq N2 N2'}
        exists {E3:eq N3 N3'}
        true.
```

This sort of theorem is known as a "uniqueness lemma" in official Twelf-speak.

To prove this theorem, we need some additional lemmas: one that says if $N_1 = N_2$ then $\mathtt{s}N_1 = \mathtt{s}N_2$ and one for the converse. These proofs are very easy to prove since the only way to create an "`eq`" relation is one that makes the variables identical:

```
%theorem
succ-deterministic :
        forall* {N1:nat} {N2:nat}
        forall {E:eq N1 N2}
        exists {F:eq (s N1) (s N2)}
        true.
%worlds () (succ-deterministic N1=N2 %{=>}% SN1=SN2).
- : succ-deterministic eq/ eq/.
%total {} (succ-deterministic _ _).

%theorem
succ-cancels :
        forall*  {N1:nat} {N2:nat}
        forall {E:eq (s N1) (s N2)}
        exists {F:eq N1 N2}
        true.
%worlds () (succ-cancels SN1=SN2 %{=>}% N1=N2).
- : succ-cancels eq/ eq/.
%total {} (succ-cancels _ _).
```

These sort of proofs might seem strange: if the proof is so trivial, why is there a need for a lemma at all? The lemma is needed because case analysis only happens at the top level. The fact that there is only one case is *global information* (because someone *could* add another case at any time).

Exercise 9 Write a proof of `plus-deterministic` using these two lemmas. □

Next, we use this theorem in our ongoing proof of `plus-associative*`, but we only get so far:

```
- : plus-associative*  X+Y=X' X'+Z=Z' Y+Z=Y' X+Y'=Z'
    <- plus-associative  X+Y=X' X'+Z=Z' Y'' Y+Z=Y'' X+Y''=Z'
    <- plus-deterministic Y+Z=Y'' Y+Z=Y' eq/ eq/ Y''=Y'
    <- ???
```

The problem is that we still have not shown than `X+Y'=Z'` even though we know `X+Y''=Z'` and `Y''=Y'`. The equality fact is not treated specially by Twelf. We need yet another lemma that says that an addition relation can be rewritten with equal variables:

```
%theorem
plus-respects-eq :
        forall* {M1} {M2} {N1} {N2} {P1} {P2}
        forall {P:plus M1 N1 P1}
```

```
               {E1:eq M1 M2} {E2:eq N1 N2} {E3:eq P1 P2}
          exists {Q:plus M2 N2 P2}
          true.
```

We will need one of these "theorems" for every relation on natural numbers, which is annoying. At least these theorems are very easy to prove since the only way to establish equality uses the same variables, as seen in `succ-deterministic` and `succ-cancels`.

Exercise 10 Write the proof of this latest lemma. □

We can use this latest lemma to finally prove our alternate associativity theorem:

```
- : plus-associative*  X+Y=X' X'+Z=Z' Y+Z=Y' X+Y'=Z'
    <- plus-associative  X+Y=X' X'+Z=Z' Y'' Y+Z=Y'' X+Y''=Z'
    <- plus-deterministic Y+Z=Y'' Y+Z=Y' eq/ eq/ Y''=Y'
    <- plus-respects-eq X+Y''=Z' eq/ Y''=Y' eq/ X+Y'=Z'.
```

The order of arguments to `plus-deterministic` is important—if we reversed the first two parameters, we get the relation `eq Y' Y''`, which can't be used directly in `plus-respects-eq`. It can however be proved that "eq" is a symmetric relation:

```
%theorem
eq-symmetric :
        forall* {M:nat} {N:nat}
        forall {E:eq M N}
        exists {F:eq N M}
        true.
```

Exercise 11 Write the (utterly trivial) proof of this theorem. □

3.3 Meta-Totality

If instead of using `plus-associative` to prove `plus-associative*`, we wanted to use `plus-associative*` to prove `plus-associative`, we have a different problem. (Of course, I'm not speaking about trying to prove each circularly using the other!) Suppose we try:

```
- : plus-associative  X+Y=X' X'+Z=Z' Y' Y+Z=Y' X+Y'=Z'
    <- plus-associative*  X+Y=X' X'+Z=Z' Y+Z=Y' X+Y'=Z'.
```

Then we get the following error:

```
Occurrence of variable Y' in input (+) argument
not necessarily ground
```

We have not bound `Y'` and indeed if we somehow bind this to something, we also have not bound `Y+Z=Y'` (an *input* parameter to the alternative associativity theorem).

But `Y'` is simply the addition of `Y` and `Z`, so perhaps one can write:

```
- : plus-associative  (X+Y=X':plus X Y X') (X'+Z=Z':plus X' Z Z')
                       Y' Y+Z=Y' X+Y'=Z'
    <- plus Y Z Y'
    <- plus-associative*  X+Y=X' X'+Z=Z' Y+Z=Y' X+Y'=Z'.
```

But this attempt falls afoul of the lack of modes for `plus`, and then the lack of proven totality for the moded addition relation. Even if these problems were fixed, we still wouldn't have a relation `Y+Z=Y'`, despite the fact that we are using it right there! Essentially we run into a level problem: the relation is an active participant in the proof, not an object. I myself don't fully understand the distinction.

In summary, totality of a relation (such as `plus`) must be proved at the meta-level (as a theorem) before it can be used at the meta-level (in a proof). Thus one writes a theorem:

```
%theorem
plus-total*: forall {N1:nat} {N2:nat}
             exists {N3:nat} {P:plus N1 N2 N3}
             true.
```

(Such a theorem is known as an "effectiveness lemma" in official Twelf-speak.) The theorem must have the number `N1` as an explicit parameter since the proof will use induction on it, and the syntax of `%total` (or `%prove`) requires the inductive parameter to be explicit. The proof of this theorem essentially repeats the definition of the relation, lifted to the meta-level:

```
%worlds () (plus-total* N1 N2 N3 N1+N2=N3).
- : plus-total* z N N plus/z.
- : plus-total* (s N1') N2 (s N3) (plus/s P)
    <- plus-total* N1' N2 N3 P.
%total N1 (plus-total* N1 _ _ _).
```

We can use an abbreviation to avoid the need to mention the numbers:

```
%abbrev plus-total = plus-total* _ _ _.
```

This abbreviation is more convenient to use, hence the asterisk at the end of the original theorem's name. Now we can write our proof of `plus-associative` as follows:

```
- : plus-associative  X+Y=X' X'+Z=Z' Y' Y+Z=Y' X+Y'=Z'
    <- plus-total Y+Z=Y'
    <- plus-associative*  X+Y=X' X'+Z=Z' Y+Z=Y' X+Y'=Z'.
```

Here `Y'` gets a binding implicitly from `Y+Z=Y'` when it is bound by `plus-total`.

3.4 Reduction

As mentioned previous, Twelf's totality checker requires that induction be structural. The structural nature of the reduction may be obscured by equality. The connection can be made available to Twelf using a `%reduces` declaration:

```
%reduces X = Y (eq X Y).
```

Similarly, we wish to able to use induction on natural numbers. To start with, we need a definition of when one natural number is greater than another:

```
gt : nat -> nat -> type.

gt/z : gt (s M) z.
gt/s : gt (s M) (s N)
    <- gt M N.
```

This definition works fine to define "greater than" but is not much use for defining reduction, because the definition doesn't show that `gt M N` always ensures that `N` is a subterm of `M`. Instead we need a definition constructed so that reduction is easily provable:

```
gt : nat -> nat -> type.

gt/1 : gt (s M) M.
gt/> : gt (s M) N
     <- gt M N.

%reduces M < N (gt N M).
```

From this new definition, it is possible for Twelf to determine that the first argument structurally includes the second.

Furthermore, as with the totality of "`plus`" before, the fact that `gt` is reductive does *not* carry over to theorems that manipulate `gt` relations. Instead, we need to lift the relation to the meta-level and prove reduction and totality at this level. (Totality is needed so that `meta-gt` can be "called" in a proof.)

```
%theorem
meta-gt : forall {M} {N} {G:gt M N}
          true.
%worlds () (meta-gt _ _ _).
- : meta-gt (s M) M (gt/1 M).
- : meta-gt (s M) N (gt/> G)
    <- meta-gt M N G.
%total M (meta-gt M _ _).
%reduces M < N (meta-gt N M _).
```

As before, we basically have to repeat the definition of the relation at the meta-level. (The theorem itself is not interesting since it has no output parameters. Totality depends merely on termination.)

3.5 Comparisons

Once we have a greater-than operation, it seems useful to be able to state the fact that given any two natural numbers, either they are equal, or one is greater than the other. As noted before, Twelf has no notion of alternation (this or that), and thus the condition must be reified in a new term family:

```
comparison : type.
less : comparison.
equal : comparison.
greater : comparison.
```

Then we define a relation that maps any pair of natural numbers to a comparison:

```
comp : nat -> nat -> comparison -> type.

comp/equal : comp z z equal.
comp/less  : comp z (s _) less.
comp/greater : comp (s _) z greater.
comp/recurse : comp (s M) (s N) C
    <- comp M N C.
```

Next, we must prove a meta-totality theorem that `comp` is total so that Twelf knows it will always return something:

```
%theorem
comp-total* : forall {M} {N}
              exists {CMP} {P:(comp M N CMP)}
              true.
%worlds () (comp-total* M N R MRN).
```

Exercise 12 Complete the proof. As usual, each proof case will basically repeat a relation case, at the meta-level. Define an abbreviation to omit all but the last parameter. □

Finally, we need theorems to indicate that if `comp` assigns equal, less or greater to a particular pair of integers, then they are `eq`, reverse `gt` or `gt`, respectively:

```
%theorem
greater-implies-gt : forall* {M} {N}
                     forall {C:comp M N greater}
                     exists {G:gt M N}
                     true.
%theorem
less-implies-lt : forall* {M} {N}
                  forall {C:comp M N less}
                  exists {G:gt N M}
                  true.
%theorem
equal-implies-eq : forall* {M} {N}
                   forall {C:comp M N equal}
                   exists {E:eq M N}
                   true.
```

In order to prove the first, we need two lemmas:

```
%theorem
succ-implies-gt-z:
          forall {M}
          exists {G:gt (s M) z}
          true.
%theorem
succ-preserves-gt:
        forall* {M} {N}
        forall {G1:gt M N}
        exists {G2:gt (s M) (s N)}
        true.
```

Exercise 13 Prove these two lemmas each by induction on their first explicit parameter. □

Exercise 14 Prove `greater-implies-gt` using the two lemmas. Then prove `less-implies-lt` similarly. Finally write the proof of `equal-implies-eq` using `succ-deterministic`. □

3.6 Contradiction

Twelf does not have a concept of negation and thus one normally doesn't prove things by contradiction. But Twelf does make it possible to reason forward from a manifest contradiction. For example, we can write a proof that if 0 equals 1, then any two numbers are equal:

```
%theorem
false-implies-eq :
        forall* {M} {N}
        forall {X:eq z (s z)}
        exists {E:eq M N}
        true.
%prove 2 {} (false-implies-eq _ _).
%worlds () (false-implies-eq _ _).
%total {} (false-implies-eq _ _).
```

Notice that this proof has no cases, which is fine, since there is no case where zero is equal to one.

If one is going to frequently use proof-by-contradiction, it is helpful to define a canonical "false" abbreviation:

```
%abbrev false = eq z (s z).
```

Then one needs a lemma for each relation that says one can generate an instance of it given this impossible relation.

Exercise 15 State and prove a theorem that "`false`" implies any addition fact one may wish. □

Normally, it's not necessary to work with contradictions because the case analysis already rules out the contradictions. However, sometimes the contradiction is not apparent immediately from the cases. In such a situation, we need to derive a contradiction such as $0 = 1$ and from there to use a theorem such as the one just "proved" to generate the required relations for the proof. For instance, case analysis makes it clear that $0 > X$ is impossible, but not that $X > Y$ contradicts $Y > X$. An inductive proof is needed to show this.

Exercise 16 Write a proof of the theorem that `gt` is "anti-symmetric," that is, if we have `gt X Y` and `gt Y X` then we can derive `false`. Hint: the induction will have no base case. □

Why is an inductive proof with no base case valid? Because it's proving a contradiction: the induction terminates, it just terminates in a contradiction.

4 Errors and Debugging

The Twelf emacs mode works pretty well at pointing you to errors. Often, however, the first error is the only one to make sense. After fixing it, you can try again (using Control-C Control-D to load just the Twelf declaration the cursor is sitting on).

Here are some of the errors I have encountered with some suggested trouble-shooting tips:

Undeclared identifier This usually means that one misspelled the name of a relation. It may also be that one tried to use a variable that doesn't start with a capital letter.

Ambiguous reconstruction This error frequently happens after one makes an earlier error. It can also happen if one neglects to state all the "parameters" to a theorem.

Ascription error This error means that the inferred type of a variable doesn't match the type one is giving it. This error message is often frustrating because Twelf inferred a type in one place and then gives an error at the ascription, rather than that at the place where the type didn't work. Sometimes the place where the error "really" is can be found by declaring the variable, rather than just using ascription.

Free variable clash When you write a pattern and give names to the variables then Twelf won't let two user-named variables to be unified. If they end up the same thing, it usually means your theorem is mixing up the variables and would not be as general as it appears in the pattern. Look to see if you can find how these two variables ended up the same. This error is often hard to find. As long as you don't have variable declarations (which would become unused if you did this), you can try ending the proof early and using binary search to find the line in the proof where the error changes from "occurrence of variable X in output (-) argument not necessarily ground" (proof is incomplete) to this error. This line often includes the erroneous step.

Argument type did not match function domain type This error is another kind of type mismatch that shows that a theorem is being called incorrectly. See comments for "ascription error" and "free variable clash" for trouble-shooting suggestions.

Type/kind constant clash This error is usually easy to fix: it means one used the wrong relation, perhaps getting the "parameters" to a theorem in the wrong order,

Left-hand side of arrow must be a type This often happens when passing too many "parameters" to a theorem.

No mode declaration for XXX If a theorem includes a relation in its proof (rather than meta-relation, as explained in the previous section) you get this error, assuming you haven't declared modes on your relation. The location of the offending relation may a variable declaration where the variable isn't used: `{T:plus X Y Z}` converts into a an "arrow type" `plus X Y Z ->` if `T` isn't used. This is an annoying aspect of Twelf's theoretical foundations: two functionally different things (variable declarations and proof steps) are easily confused by the system.

Occurrence of variable X in output (-) argument not necessarily ground This error means that the proof is incomplete; it doesn't show the existence of this variable in the theorem's "output." This variable is usually part of a relation that hasn't been produced by the proof. One reason for the error could be that the relation produced uses different variables than the one required for the proof.

Occurrence of variable X in input (+) argument not necessarily ground In your proof, you call other "theorems." This error happens when the theorem's requirements (inputs) are not satisified. You may need to explicitly state the existence of a relation by "calling" a meta-totality theorem.

Occurrence of variable X in output (-) argument not free This error also happens when you use a theorem incorrectly in your proof. Unlike the previous case, this error means that you are assuming too much about the *results* of the theorem. For example, the theorem guarantees the existence of some number, but your proof is assuming that the result is (say) non-zero. You may need to factor out a lemma to do case analysis on the result of the theorem.

Definition violation This error happens sporadically when doing line-by-line (Control-C Control-D) interactions. Try loading the whole file instead. It also happens when you write a theorem or relation abbreviation and don't declare it as an abbreviation. I consider the latter behavior to be a bug in Twelf: the definition can't affect later theorems.

5 Advanced Topics

In this section, I briefly mention some more advanced topics. Because I myself have not had experience in these areas, I have little to say. The reader should consult the resources mentioned at the start of this document.

Twelf has the ability to check mutually inductive theorems—multiple call patterns can be specified in a `%total` declaration.

One of interesting aspects of Twelf is the ability to dynamically add relations during a "call" to another relation. Dynamic additions can be used in many places where one might use "assert" and "retract" in Prolog.

Higher-order syntax, in which function terms are represented by functions at the Twelf level avoids the complications of explicit environment objects, but requires "regular contexts" (non-trivial worlds).

6 Conclusions

Twelf is well-suited for writing typing proofs but the user needs some adjusting to its various idiosyncracies. There are number of notable shortcomings (especially the fact that the theorem prover does not produce a proof) that should and will be addressed as soon as possible. There are a number of other improvements (such as polymorphism, and the ability to use totality or reduction information at the meta-level) that would make Twelf more convenient to use.