Twelf User's Guide

Version 1.3

Frank Pfenning and Carsten Schuermann

1 Introduction

Twelf is the current version of a succession of implementations of the logical framework LF. Previous systems include Elf (which provided type reconstruction and the operational semantics reimplemented in Twelf) and MLF (which implemented module-level constructs loosely based on the signatures and functors of ML still missing from Twelf).

Twelf should be understood as research software. This means comments, suggestions, and bug reports are extremely welcome, but there are no guarantees regarding response times. The same remark applies to these notes which constitute the only documentation on the present Twelf implementation.

For current information including download instructions, publications, and mailing list, see the Twelf home page at `http://www.cs.cmu.edu/~twelf/`. This User's Guide is published as

> Frank Pfenning and Carsten Schuermann
> *Twelf User's Guide*
> Technical Report CMU-CS-98-173, Department of Computer Science,
> Carnegie Mellon University, November 1998.

Below we state the typographic conventions in this manual.

`code` for Twelf or ML code

'`samp`' for characters and small code fragments

metavar for placeholders in code

`keyboard` for input in verbatim examples

`KEY` for keystrokes

math for mathematical expressions

emph for emphasized phrases

File names for examples given in this guide are relative to the main directory of the Twelf installation. For example '`examples/guide/nd.elf`' may be found in '`/usr/local/twelf/examples/guide/nd.elf`' if Twelf was installed into the '`/usr/local/`' directory.

1.1 New Features

The current version 1.3 from September 13, 2000 incorporates the following major changes from Twelf 1.2 from August 27, 1998.

Constraints (see Chapter 6 [Constraint Domains], page 27).
: Type reconstruction and the logic programming engine (but not yet the theorem prover) allow various constraint domains in the style of constraint logic programming languages. The main ones are equalities and inequalities over rationals and integers.

Tracing and Breakpoints (see Section 10.5 [Tracing and Breakpoints], page 60)
: The logic programming engine now support tracing and setting of breakpoints for illustration and debugging purposes.

Theorem Prover (see Chapter 9 [Theorem Prover], page 49)
: The theorem prover now allows quantification over regular context. The theorem prover will also use previously proved theorems with `%prove`) and ignore those with `%establish`, which is otherwise equivalent. In unsafe mode, `%assert` can be used to claim theorems. However, at present no longer generates proof terms.

Reduction Checking (see Section 8.2 [Reduction Declaration], page 44)
: The termination checker has been extended to verify if output arguments to a predicate are smaller than some inputs with the `%reduces` declaration.

Signature Printing (see Section 10.4 [Signature Printing], page 60)
: Signatures can now be printed, also in TeX format.

Abbreviations (see Section 3.3 [Definitions], page 9)
: Added abbreviations (`%abbrev`) which, unlike definition, do not need to be strict.

Name Preferences (see Section 3.5 [Name Preferences], page 11)
: Name preference declarations (`%name`) now allow an optional second argument for naming of parameters.

1.2 Quick Start

Assuming you are running on a Unix system with SML of New Jersey already installed (see Chapter 13 [Installation], page 81) you can build Twelf as follows. Here '`%`' is assumed to be the shell prompt. You may need to edit the file '`Makefile`' to give the proper location for `sml-cm`.

```
% gunzip twelf-1-3.tar.gz
% tar -xf twelf-1-3.tar
% cd twelf
% make
% bin/twelf-server
Twelf 1.3, Sep 13 2000
%% OK %%
```

You can now load the examples used in this guide and pose an example query as shown below. The prompt from the Twelf top-level is '`?-`'. To drop from the Twelf top-level to the ML top-level, type `C-c` (`CTRL c`). To exit the Twelf server you may issue the `quit` command or type `C-d` (`CTRL c`).

```
Config.read examples/guide/sources.cfg
Config.load
top
?- of (lam [x] x) T.
Solving...
T = arrow T1 T1.
More? y
No more solutions
?- C-c
interrupt
%% OK %%
quit
%
```

2 Lexical Conventions

Lexical analysis of Twelf has purposely been kept simple, with few reserved characters and identifiers. As a result one may need to use more whitespace to separate identifiers than in other languages. For example, '`A->B`' or '`A+B`' are single identifiers, while '`A -> B`' and '`A + B`' both consist of 3 identifiers.

During parsing, identifiers are resolved as reserved identifiers, constants, bound variables, or free variables, following the usual rules of static scoping in lambda-calculi.

2.1 Reserved Characters

The following table lists the reserved characters in Twelf.

'`:`' colon, constant declaration or ascription

'`.`' period, terminates declarations

'`(`' '`)`' parentheses, for grouping terms

'`[`' '`]`' brackets, for lambda abstraction

'`{`' '`}`' braces, for quantification (dependent function types)

whitespace
separates identifiers; one of space, newline, tab, carriage return, vertical tab or formfeed

'`%`' introduces comments or special keyword declarations

'`%`*whitespace*' '`%%`'
comment terminated by the end of the line, may contain any characters

'`%{`' '`}%`' delimited comment, nested '`%{`' and '`}%`' must match

'`%`*keyword*'
various declarations

'`%.`' end of input stream

'`"`' doublequote, disallowed

other printing characters
identifier constituents

2.2 Identifiers

All printing characters that are not reserved can be included in identifiers, which are separated by whitespace or reserved characters. In particular, `A->B` is an identifier, whereas `A -> B` stands for the type of functions from `A` to `B`.

An uppercase identifier is one which begins with an underscore '`_`' or a letter in the range '`A`' through '`Z`'. A lowercase identifier begins with any other character except a reserved one. Numbers also count as lowercase identifiers and are not interpreted specially. Free variables in a declaration must be uppercase, bound variables and constants may be either uppercase or lowercase identifiers.

There are also four reserved identifiers with a predefined meaning which cannot be changed. Keep in mind that these can be constituents of other identifers which are not interpreted specially.

'`->`' function type

'`<-`' reverse function type

'`_`' hole, to be filled by term reconstruction

'`=`' definition

'`type`' the kind *type*

Constants have static scope, which means that they can be shadowed by subsequent declarations. A shadowed identifier (which can no longer be referred to in input) is printed as `%`*id*`%`. The printer for terms renames bound variables so they do not shadow constants.

Free uppercase identifiers in declarations represent schematic variables. In order to distinguish them from other kinds of variables and constants they are printed as '`'`*id*' (backquote, followed by the identifer name) in error messages.

3 Syntax

In LF, deductive systems are represented by signatures consisting of constant declarations. Twelf implements declarations in a straightforward way and generalizes signatures by also allowing definitions, which are semantically transparent. Twelf currently does not have module-level constructs so that, for example, signatures cannot be named. Instead, multiple signatures can be manipulated in the programming environment using configurations (see Section 10.1 [Configurations], page 57).

The LF type theory which underlies LF is stratified into three levels: objects M and N, types A and B, and kinds K. Twelf does not syntactically distinguish these levels and simply uses one syntactic category of term. Similarly, object-level constants c and type-level constants a as well as variables share one name space of identifiers.

In explanations and examples we will use letters following the mathematical conventions above to clarify the roles of various terms. We also use U and V to stand for arbitrary terms.

3.1 Grammar

The grammar below defines the non-terminals `sig`, `decl`, `term` and uses the terminal `id` which stands for identifers (see Section 2.2 [Identifiers], page 6). The comments show the meaning in LF. There are various special declarations `%`*keyword* such as `%infix` or `%theorem` which are omitted here and detailed in the appropriate sections.

```
sig  ::=                        % Empty signature
       | decl sig               % Constant declaration

decl ::= id : term.             % a : K  or  c : A
       | id : term = term.      % d : A = M
       | id = term.             % d = M
       | _ : term = term.       % anonymous definition, for type-checking
       | _ = term.              % anonymous definition, for type-checking
       | %abbrev adecl.         % abbreviation
       | %infix ixdecl.         % operator declaration
       | %prefix pxdecl.        % operator declaration
       | %postfix pxdecl.       % operator declaration
       | %name namepref.        % name preference declaration
       | %query qdecl.          % query declaration
       | %solve id : term.      % solve declaration
       | %mode mdecl.           % mode declaration
       | %terminates tdecl.     % termination declaration
       | %reduces rdecl.        % reduction declaration
       | %theorem thdecl.       % theorem declaration
       | %prove pdecl.          % prove declaration
       | %establish pdecl.      % prove declaration, do not use as lemma later
       | %assert callpats.      % assert theorem (requires Twelf.unsafe)
       | %use domain.           % installs constraint domain

term ::= type                   % type
       | id                     % variable x or constant a or c
       | term -> term           % A -> B
       | term <- term           % A <- B, same as B -> A
       | {id : term} term       % Pi x : A. K  or  Pi x : A. B
       | [id : term] term       % lambda x : A. B  or  lambda x : A. M
       | term term              % A M  or  M N
       | term : term            % explicit type ascription
       | _                      % hole, to be filled by term reconstruction
       | {id} term              % same as {id:_} term
       | [id] term              % same as [id:_] term
```

The constructs `{x:U} V` and `[x:U] V` bind the identifier `x` in `V`, which may shadow other constants or bound variables. As usual in type theory, `U -> V` is treated as an abbreviation for `{x:U} V` where `x` does not appear in `V`. However, there is a subtlety in that the latter allows an implicit argument (see Section 4.2 [Implicit Arguments], page 14) to depend on `x` while the former does not.

In the order of precedence, we disambiguate the syntax as follows:

1. Juxtaposition (application) is left associative and has highest precedence.
2. User declared infix, prefix, or postfix operators (see below).
3. '`->`' is right and '`<-`' left associative with equal precedence.

4. ‘:’ is left associative.
5. ‘{}’ and ‘[]’ are weak prefix operators.

For example, the following are parsed identically:

```
d : a <- b <- {x} c x -> p x.
d : ({x} c x -> p x) -> b -> a.
d : ((a <- b) <- ({x:_} ((c x) -> (p x)))).
```

3.2 Constructor Declaration

New type families or object constructors can be introduced with

```
condec ::= id : term.      % a : K or c : A
```

Here `a` stands for a type family and `K` for its kind, whereas `c` is an objects constructor and `A` its type. Identifiers are resolved as follows:

1. Any identifier `x` may be bound by the innermost enclosing binder for `x` of the form `{x:A}` or `[x:A]`.
2. Any identifer which is not explicitly bound may be a declared or defined constant.
3. Any uppercase identifier, that is, identifier starting with ‘_’ (underscore) or an upper case letter, may be a free variable. Free variables are interpreted universally and their type is inferred from their occurrences (see Chapter 4 [Term Reconstruction], page 13).
4. Any other undeclared identifier is flagged as an error.

3.3 Definitions

Twelf supports notational definitions and abbreviations. Semantically, both are completely transparent, that is, both for type checking and the operational semantics, definitions may be expanded.

```
adecl ::= id : term = term          % d : A = M
        | id = term                 % d = M

defn ::= adecl.                     % definition
       | %abbrev adecl.             % abbreviation
```

where the second form of declaration is equivalent to `id : _ = term`. Definitions or abbreviations can only be made on the level of objects, not at the level of type families because the interaction of such definitions with logic programming search has not been fully investigated.

In order to avoid the expansion of defined constants as much as possible, declarations `id : term = term.` must be strict (see Section 4.4 [Strict Definitions], page 16). A definition of a constant `c` is strict if all arguments to `c` (implicit or explicit) have at least one strict occurrence (see Section 4.3 [Strict Occurrences], page 15) in the right-hand side of the definition, and the right-hand side contains at least one constant. In practice, most notational definitions are strict. For some examples, see Section 3.6 [Sample Signature], page 11 and Section 4.4 [Strict Definitions], page 16. Twelf tries to preserve strict definitions as much as possible, instead of expanding them.

The `%abbrev` declaration defines an *abbreviation* which need not be strict. However, it will be expanded immediately upon parsing and will not be used in output.

The power of definitions in Twelf, however, is severely limited by the lack of recursion. It should only be thought of as notational definition, not as a computational mechanism. Complex operations need to be defined as logic programs, taking advantage of the operational semantics assigned to signatures (see Chapter 5 [Logic Programming], page 19).

3.4 Operator Declaration

The user may declare constants to be infix, prefix, or postfix operators. Operator precedence properties are associated with constants, which must therefore already have been declared with a type or kind and a possible definition. It is illegal to shadow an infix, prefix, or postfix operator with a bound variable. We use `nat` for the terminal natural numbers.

```
assoc ::= none   % not associative
        | left   % left associative
        | right  % right associative

prec ::= nat     % 0 <= prec < 10000

ixdecl ::= assoc prec id

pxdecl ::= prec id

decl ::= ...
       | %infix ixdecl.
       | %prefix pxdecl.
       | %postfix pxdecl.
```

During parsing, ambiguous successive operators of identical precedence such as `a <- b -> c` are flagged as errors. Note that it is not possible to declare an operator with equal or higher precedence than juxtaposition or equal or lower precedence than '`->`' and '`<-`'.

3.5 Name Preferences

During printing, Twelf frequently has to assign names to anonymous variables. In order to improve readability, the user can declare a name preference for anonymous variables based on their type. Thus name preferences are declared for type family constants. Note that name preferences are not used to disambiguate the types of identifiers during parsing.

```
namepref ::= id          % existential variables
           | id id       % existential variables, parameters

decl ::= ...
       | %name id namepref.
```

Following our same conventions, a name preference declaration has the form `%name` *a id*, that is, the first identifier must be a type family already declared and the second is the name preference for variables of type *a*. The second identifier must be uppercase, that is, start with a letter from '`A`' through '`Z`' or an underscore '`_`'. Anonymous variables will then be named *id*1, *id*2, etc.

In the second form, we can give a separate name preference for free (existential) variables and parameters. The second one will typicall be lowercase, as in `%name exp E x`.

3.6 Sample Signature

Below is a signature for intuitionistic first-order logic over an unspecified domain of individuals and atomic propositions. It illustrates constant declarations and definitions and the use of operator precedence and name preference declarations. It may be found in the file '`examples/guide/nd.elf`'.

```
%%% Individuals
i : type.                    %name i T

%%% Propositions
o : type.                    %name o A

imp    : o -> o -> o.        %infix right 10 imp
and    : o -> o -> o.        %infix right 11 and
true   : o.
or     : o -> o -> o.        %infix right 11 or
false  : o.
forall : (i -> o) -> o.
exists : (i -> o) -> o.

not : o -> o = [A:o] A imp false.

%%% Natural Deductions

nd : o -> type.              %name nd D

impi    : (nd A -> nd B) -> nd (A imp B).
impe    : nd (A imp B) -> nd A -> nd B.
andi    : nd A -> nd B -> nd (A and B).
ande1   : nd (A and B) -> nd A.
ande2   : nd (A and B) -> nd B.
truei   : nd (true).
% no truee
ori1    : nd A -> nd (A or B).
ori2    : nd B -> nd (A or B).
ore     : nd (A or B) -> (nd A -> nd C) -> (nd B -> nd C) -> nd C.
% no falsei
falsee  : nd false -> nd C.
foralli : ({x:i} nd (A x)) -> nd (forall A).
foralle : nd (forall A) -> {T:i} nd (A T).
existsi : {T:i} nd (A T) -> nd (exists A).
existse : nd (exists A) -> ({x:i} nd (A x) -> nd C) -> nd C.

noti : (nd A -> nd false) -> nd (not A)
     = [D:nd A -> nd false] impi D.
note : nd (not A) -> nd A -> nd false
     = [D:nd (not A)] [E:nd A] impe D E.
```

4 Term Reconstruction

Representations of deductions in LF typically contain a lot of redundant information. In order to make LF practical, Twelf gives the user the opportunity to omit redundant information in declarations and reconstructs it from context. Unlike for functional languages, this requires recovering objects as well as types, so we refer to this phase as term reconstruction.

There are criteria which guarantee that the term reconstruction problem is decidable, but unfortunately these criteria are either very complicated or still force much redundant information to be supplied. Therefore, the Twelf implementation employs a reconstruction algorithm which always terminates and gives one of three answers:

1. yes, and here is the most general reconstruction;
2. no, and here is the problem; or
3. maybe.

The last characterizes the situations where there is insufficient information to guarantee a most general solution to the term reconstruction problem. Because of the decidable nature of type-checking in LF, the user can always annotate the term further until it falls into one of the definitive categories.

4.1 Implicit Quantifiers

The model of term reconstruction employed by Twelf is straightforward, although it employs a relatively complex algorithm. The basic principle is a duality between quantifiers omitted in a constant declaration and implicit arguments where the constant is used. Recall some definitions in the signature defining natural deductions (see Section 3.6 [Sample Signature], page 11).

```
o : type.
and : o -> o -> o.   %infix right 10 and
nd : o -> type.
andi : nd A -> nd B -> nd (A and B).
```

The last declaration contains `A` and `B` as free variables. Type reconstruction infers most general types for the free variables in a constant declaration and adds implicit quantifiers. In the example above, `A` and `B` must both be of type `o`. The internal form of the constant thus has one of the following two forms.

```
andi : {A:o} {B:o} nd A -> nd B -> nd (A and B).
andi : {B:o} {A:o} nd A -> nd B -> nd (A and B).
```

These forms are printed during type reconstruction, so the user can examine if the result of reconstruction matches his expectations.

4.2 Implicit Arguments

The quantifiers on `A` and `B` in the declaration

```
andi : nd A -> nd B -> nd (A and B).
```

were implicit. The corresponding arguments to `andi` are also implicit. In fact, since the order of the reconstructed quantifiers is arbitrary, we cannot know in which order to supply the arguments, so they must always be omitted. Thus a constant with n implicit quantifiers is supplied with n implicit arguments whereever it is seen. These implicit arguments are existential variables whose value may be determined from context by unification.

For example, using also

```
true : o.
truei: nd (true).
```

we have

```
(andi truei truei) : nd (true and true).
```

During parsing, the expression `(andi truei truei)` is interpreted as

```
(andi _ _ truei truei)
```

where the two underscores stand for the implicit `A` and `B` arguments to `andi`. They are replaced by existential variables whose value will be determined during type reconstruction. We call them `A1` and `A2` and reason as follows.

```
|- andi : {A:o} {B:o} nd A -> nd B -> nd (A and B)
|- andi A1 : {B:o} nd A1 -> nd B -> nd (A1 and B)
|- andi A1 A2 : nd A1 -> nd A2 -> nd (A1 and A2)
```

At this point, we need a to infer the type of the application `(andi A1 A2) truei`. This equates the actual type of the argument with the expected type of the argument.

```
|- andi A1 A2 : nd A1 -> nd A2 -> nd (A1 and A2)
|- truei : nd true
------------------------------------------------
|- andi A1 A2 truei : nd A2 -> nd (A1 and A2)
   where nd true = nd A1
```

The equation can be solved by instantiating `A1` to `true` and we continue:

```
|- andi true A2 truei : nd A2 -> nd (true and A2)
|- truei : nd true
------------------------------------------------
|- andi true A2 truei truei : nd (true and A2)
   where nd true = nd A2
|- andi true true truei truei : nd (true and true)
```

The last line is the expected result. In this way, term reconstruction can always be reduced to solving equations such that every solution to the set of equations leads to a valid typing and vice versa.

4.3 Strict Occurrences

Both for type reconstruction and the operational semantics, Twelf must solve equations between objects and types. Unfortunately, it is undecidable if a set of equations in the LF type theory has a solution. Worse yet, even if it has solutions, it may not have a most general solution. Therefore, Twelf postpones difficult equations as constraints and solves only those within the pattern fragment (see *Miller 1991, Journal of Logic and Compuation* and *Pfenning 1991, Logical Frameworks*). In this fragment, principal solutions always exist and can be computed efficiently. If constraints remain after term reconstruction, the constant declaration is rejected as ambiguous which indicates that the user must supply more type information. We illustrate this phenomenon and a typical solution in our natural deduction example.

A central concept useful for understanding the finer details of type reconstruction is the notion of a *strict occurrence* of a free variable. We call a position in a term *rigid* if it is not in the argument of a free variable. We then call an occurrence of a free variable *strict* if the occurrence is in a rigid position and all its arguments (possibly none) are distinct bound variables.

If all free variable occurrences in all declarations in a signature are strict, then term reconstruction will always either fail or succeed with a principal solution, provided no further terms are omitted (that is, replaced by an underscore).

If a free variable in a declaration of a constant `c` has no strict occurrence at all, then its type can almost never be inferred and most uses of `c` will lead to a constraint.

If a free variable has strict and non-strict occurrences then in most cases term reconstruction will provide a definitive answer, but there is no guarantee. Mostly this is because most general answers simply do not exist, but sometimes because the algorithm generates, but cannot solve constraints with unique solutions.

We use some advanced examples from the natural deduction signature to illustrate these concepts and ideas. In the declarations

```
forall i : ({x:i} nd (A x)) -> nd (forall A).
foralle : nd (forall A) -> {T:i} nd (A T).
```

all free variables have a strict occurrence. However, if we had decided to leave `T` as an implicit argument,

```
foralle : nd (forall A) -> nd (A T).
```

then `T` has no strict occurrence. While this declaration is accepted as unambiguous (with `A:i -> o` and `T:i`), any future use of `foralle` most likely leads to constraints on `T` which cannot be solved.

4.4 Strict Definitions

Definitions are currently restricted so that each argument to the defined constant, may it be implicit or explicit, must have at least one strict occurrence in the right-hand side. For example, the definition of `not` in the signature for natural deduction (see Section 3.6 [Sample Signature], page 11)

```
not : o -> o = [A:o] A imp false.
```

is strict since the only argument `A` has a strict occurrence in `A imp false`. On the other hand, the definition

```
noti : ({p:o} nd A -> nd p) -> nd (not A)
     = [D] impi ([u:nd A] D false u).
```

which gives a possible derived introduction rule for negation is not strict: the argument `D` has only one occurrence, and this occurrence is not strict since the argument `false` is not a variable bound in the body, but a constant.

However, the definitions

```
noti : (nd A -> nd false) -> nd (not A)
     = [D:nd A -> nd false] impi D.
note : nd (not A) -> nd A -> nd false
     = [D:nd (not A)] [E:nd A] impe D E.
```

are both strict since arguments `D` and `E` both have strict occurrences. Type-checking these definitions requires that the definition of `not A` is expanded to `A imp false`.

Note that free variables in the type and the right-hand side of a definition are shared. In the above example, `A` occurs both in the types and the right hand side and it should be thought of as the same `A`. With the implicit quantifiers and abstractions restored, the definitions above have the following form.

```
noti : {A:o} (nd A -> nd false) -> nd (not A)
   = [A:o] [D:nd A -> nd false] impi D.
note : {A:o} nd (not A) -> nd A -> nd false
   = [A:o] [D:nd (not A)] [E:nd A] impe D E.
```

4.5 Type Ascription

In some circumstances it is useful to directly ascribe a type in order to disambiguate declarations. For example, the term `oril truei` has principal type `nd (true or B)` for a free variable `B`. If we want to constrain this to a derivation of `nd (true or false)` we can write `oril truei : nd (true or false)`.

Explicit type ascription sometimes helps when the source of a type error is particularly hard to discern: we can ascribe an expected type to a subterm, thus verifying our intuition about constituent terms in a declaration.

4.6 Error Messages

When term reconstruction fails, Twelf issues an error message with the location of the declaration in which the problem occurred and the disagreement encountered. A typical message is

```
examples/nd/nd.elf:37.35-37.41 Error: Type mismatch
Expected: o
Found:    (i -> o) -> o
Expression clash
```

which points to an error in the file '`examples/nd/nd.elf`', line 37, characters 35 through 41 where an argument to a function was expected to have type `o`, but was found to have type `(i -> o) -> o`.

If constraints remain, the error location is the whole declaration with the message

filename`:`*location* `Error: Typing ambiguous -- unresolved constraints`

The *filename* and *location* information can be used by Emacs (see Chapter 12 [Emacs Interface], page 69) to jump to the specified location in the given file for editing of the incorrect declaration for the constant `c`. The *location* has the form *line1*`.`*column1*`-`*line2*`.`*column2* and represent Twelf's best guess as to the source of the error. Due to the propagation of non-trivial constraints the source of a type reconstruction failure can sometimes not be pinpointed very precisely.

5 Logic Programming

Twelf gives an operational interpretation to signatures under the computation-as-proof-search paradigm. The fundamental idea is to fix a simple search strategy and then search for a derivation of a query according to this strategy. The result may be a substitution for the free variables in a query and a derivation, or explicit failure. It is also possible that the computation does not terminate.

A query can be posed in three different ways: as a `%query` declaration, as a `%solve` declaration, or interactively, using a top-level invoked from ML with `Twelf.top` which prompts with '`?-`' (see Section 5.3 [Interactive Queries], page 20).

```
query ::= id : term    % X : A, X a free variable
        | term         % A

bound ::= nat          % number of solutions
        | *            % unbounded number

qdecl ::= bound bound query % expected solutions, try limit, query

decl ::= ...
       | %query qdecl.       % execute query
       | %solve id : term.   % solve and name proof term
```

In all of these cases, the free variables in a query are interpreted existentially, which is in contrast to constant declarations where free variables are interpreted universally. In particular, free variables might be instantiated during type reconstruction and during execution of the query.

5.1 Query Declaration

The query form

`%query` *expected try* `A.`

will try to solve the query `A` and verify that it gives the *expected* number of solutions, but it will never try to find more than indicated by *try*. It succeeds and prints a message, whose precise form depends on the value of `Twelf.chatter` if `A` has the expected number of solutions; otherwise it either fails with an error message or does not terminate. '`%query`' has no other effect on the state of Twelf. Here are some examples.

```
%query 1 * A.        % check that A has exactly one solution
%query 1 1 A.        % check that A has at least one solution
%query * 3 A.        % A has infinitely many solutions, check 3
%query * * A.        % fails if A has finitely many solutions
%query 1 0 A.        % skip this query
```

5.2 Solve Declaration

The query form

```
%solve c : A.
```

will search for the first solution `M` of `A` and then define

```
c : A = M.
```

If there are any free variables remaining in `M` or `A` after search, they will be implicitly quantified in the new definition. This form of definition is particularly useful to compute and name inputs to future queries. An example of this feature from the file '`examples/nd/lam.elf`' can be found in Section 9.5 [Proof Realizations], page 54.

5.3 Interactive Queries

An interactive top-level can be invoked using the SML expression `Twelf.top ();`. The top-level prompts with '`?- `' and awaits the input of a query, terminated by a period '`.`' and a `RET`.

After the query has been parsed, Twelf reconstructs implicit type information, issuing a warning if constraints remain. The result is executed as a query. At any point during the processing of a query the user may interrupt with `C-c` (that is, `CTRL` and `c`) to drop back into ML's interactive top-level.

When Twelf has found a solution, it prints the *answer substitution* for all free variables in the query, including the proof term variable if one was given. It also notes if there are remaining equational constraints, but currently does not print them.

The top-level then waits for input, which is interpreted as follows `y`, `Y`, or `;` backtrack and search for another solution

`q` or `Q` quit Twelf's top-level and return to ML

`n`, `N`, or anything else
return to prompt for another query

5.4 Sample Trace

As an example we consider lists of propositions and some simple operations on them, as they might be used when programming a theorem prover.

```
list : type.
nil  : list.
cons : o -> list -> list.
```

First, we want to write a program to append two lists to obtain their concatenation. This is expressed as a relation between the three lists, which in turn is implemented as a type family

```
append  : list -> list -> list -> type.

appNil  : append nil K K.
appCons : append (cons X L) K (cons X M)
           <- append L K M.
```

Here, we use the synonym `A <- B` for `B -> A` to improve readability. We say `A` *if* `B`.

The first sample query concatenates the singleton lists containing `true` and `false`. We proceed as if we had loaded the appropriate files and started a top-level with `Twelf.top ();`.

```
?- append (cons true nil) (cons false nil) L.
```

Here, `L` is a free existential variable. We search for an object `M : append (cons true nil) (cons false nil) L`, even though this object will not be shown in this form or query. Each constant declaration in the signature is tried in turn, unifying the head with the goal above. In this manner, we obtain the following sequence of goals and actions. Note that the intermediate forms and comments are not printed when this is run. They are added here to illustrate the behavior.

```
% original goal after parsing and type reconstruction
?- append (cons true nil) (cons false nil) L.
[try appNil:
     append nil K1 K1
   = append (cons true nil) (cons false nil) L
 unification fails with constant clash: nil <> cons
]
[try appCons:
     append (cons X1 L1) K2 (cons X1 M1)
   = append (cons true nil) (cons false nil) L
 unification succeeds with
 X1 = true, L1 = nil, K2 = cons false nil, L = cons true M1
]
% subgoal
?- append nil (cons false nil) M1.
[try appNil:
     append nil K3 K3
   = append nil (cons false nil) M1
 unification and subgoal succeeds with
 K3 = cons false nil, M1 = cons false nil
]
```

At this point the overall goal succeeds and we read off the answer substitution for the only free variable in the query, namely `L`. It was first determined to be `cons true M1` and then `M1` was instantiated to `cons false nil`, leading to the instantiation

```
L = cons true (cons false nil).
```

If instead we pose the query

```
?- X : append (cons true nil) (cons false nil) L.
```

we also obtain the proof term

```
L = cons true (cons false nil);
X = appCons appNil.
```

As another example we consider a query with several solutions which are enumerated when we ask for further results. This time we do not trace the steps of the execution, but show the interaction verbatim.

```
?- append L K (cons true (cons false nil)).
Solving...
K = cons true (cons false nil);
L = nil.
More? y
K = cons false nil;
L = cons true nil.
More? y
K = nil;
L = cons true (cons false nil).
More? y
No more solutions
```

5.5 Operational Semantics

The operational semantics of Twelf is a form of typed constraint logic programming. We will use standard terminology from this area. A type family which is used in a program or goal is called a *predicate*. A constant declaration in a signature which is available during search is called a *clause*. A clause typically has the form `c : a M1 ... Mm <- A1 <- ... <- An`, where `a M1 ... Mm` is the *head of the clause* and `A1` through `An` are the *subgoals*. A clause is used to reduce a goal to subgoals by a process called *backchaining*. Backchaining unifies the head of the clause with the current goal to generate *subgoals*. Next, we *select* one of the subgoals as a current goal and continue the search process. Actually, instead of unification (which is undecidable in LF), Twelf employs *constraint simplification* and carries along equational constraints in a normal form.

A hypothesis which is introduced during search is a *local assumption*; a parameter is a *local parameter*. Parameters act like constants in unification, except that their occurrences might be restricted due to *parameter dependency*.

Without going into a formal description, here are the central ideas of the operational semantics.

Clause selection.
: The clauses are tried in the following order: from most recent to least recent local assumption, then from first to last clause in the global signature.

Subgoal selection.
: Subgoals are solved from the inside out. For example, when a clause `c : A <- B <- C.` is applied to solve the goal `?- A.` then the first subgoal is `B` and the second subgoal `C`. Truly dependent variables will only be subject to unification and never give rise to a subgoal. For example `c : {X:b} a X <- a c` is a clause with head `a X`, subgoal `a c`, and existential variable `X`.

Unification.

An atomic goal is unified with the clause head using higher-order pattern unification. All equations outside this fragment are postponed and carried along as constraints.

Local assumptions.

A goal of the form `?- A -> B.` introduces a local assumption `A` and then solves `B` under this assumption. To solve atomic goals, local assumptions are tried before global clauses, using the most recently made assumption first. Note that this is different from Prolog `assert` in that `A` is available only for solving `B`.

Local parameters.

Parameters are introduced into proof search by goals of the form `?- {x:A} B.` which generates a *new* parameter `a` and then solves the result of substituting `a` for `x` in `B`. Parameters are also called *universal variables* since they are not subject to instantiation during unification. Local parameters will never be used as local assumptions during search.

5.6 Sample Program

As an example, we consider simple type inference for the pure lambda-calculus. An extension of this example to Mini-ML is given in the course notes *Pfenning 1992, Computation and Deduction*. The code below can be found in the file '`examples/guide/lam.elf`'.

```
% Simple types
tp : type.                                  %name tp T.

arrow : tp -> tp -> tp.                     % T1 => T2

% Expressions
exp : type.                                 %name exp E.

lam   : (exp -> exp) -> exp.                % lam x. E
app   : exp -> exp -> exp.                  % (E1 E2)

% Type inference
% |- E : T  (expression E has type T)

of : exp -> tp -> type.                     %name of P.

tp_lam : of (lam E) (arrow T1 T2)           % |- lam x. E : T1 => T2
          <- ({x:exp}                       % if  x:T1 |- E : T2.
                of x T1 -> of (E x) T2).

tp_app : of (app E1 E2) T1                  % |- E1 E2 : T1
          <- of E1 (arrow T2 T1)            % if  |- E1 : T2 => T1
          <- of E2 T2.                      % and |- E2 : T2.
```

Again, we have used the notation `A <- B` to emphasize the interpretation of constant declarations as clauses. We now trace the query which infers the most general type of the identity function, represented as `lam [x:exp] x`. We indicate the scope of hypotheses which are introduced during search by indentation.

```
% original query, T free
?- of (lam [x:exp] x) T.
% use tp_lam with E = ([x:exp] x) and T = arrow T1 T2
% subgoal
?- {x:exp} of x T1 -> of x T2.
% introduce parameter e
?- of e T1 -> of e T2.
% introduce hypothesis labelled p
p:of e T1
 ?- of e T2.
 % succeed by hypothesis p with T1 = T2
```

At this point the query succeeds and prints the answer substitution.

```
T = arrow T1 T1.
More? y
No more solutions
```

We requested more solution by typing `y`, but there are no further possibilities. The free variable `T1` in the answer substitution means that every instance of `arrow T1 T1` provides a solution to the original query. In other words, `lam [x:exp] x` has type `arrow T1 T1` for all types `T1`.

As a second example we verify that self-application is not well-typed in the simply-typed lambda-calculus.

```
?- of (lam [x:exp] app x x) T.
% use tp_lam with T = arrow T1 T2
% subgoal
?- {x:exp} of x T1 -> of (app x x) T2.
% introduce parameter e
?- of e T1 -> of (app e e) T2.
% introduce hypothesis p:of a T1
p:of e T1
 ?- of (app e e) T2.
 % use tp_app
 % first subgoal
 ?- of e (arrow T3 T2).
 % succeed by hypothesis p with T1 = arrow T3 T2
 % second subgoal
 ?- of e T3.
 % fail, since T3 = arrow T3 T2 has no solution
```

At the point where the second subgoals fails we backtrack. However, all other alternatives fail immediately, since the clause head does not unify with the goal and the overall query fails.

6 Constraint Domains

Constraints-based extensions are highly experimental and under active development. Use of extensions other than the ones pertaining rational arithmetic is strongly discouraged. We refer to extension by constraint domains as Twelf(X), where X refers different domains.

Twelf(X) extensions allow Twelf to deal more efficiently with some specific domains (e.g. numbers). They do so by

1. modifying type reconstruction to accommodate other equivalences beyond those entailed by traditional beta-eta-conversion;
2. defining special types, on which proof search is performed using ad-hoc decision procedures rather than traditional depth-first strategy;
3. adding countably many new symbols to the language, signifying all the elements of the domain.

6.1 Installing an Extension

Extensions are installed using the declaration

`%use` *extension name*

For example,

```
%use equality/rationals.
```

loads the extension dealing with equality over the rationals. Typically, an extension introduces new symbols in the signature. For example, `equality/rationals` adds a type for rational numbers:

```
rational : type.
```

In addition to these symbols, a countably infinite set of "special" ones may be also accepted by the system as the result of loading one extension. In our example, after loading `equality/rationals` the symbols

```
135  5/13  ~1/4
```

become valid constants of type `rational`.

Finally, it is possible for an extension to be built on top of others, and therefore to depend on them. Hence, loading an extension usually causes all the others it depends upon to be loaded as well, if they have not been already. For example, the extension `inequality/rationals` requires `equality/rationals`, and hence the declaration

```
%use inequality/rationals.
```

subsumes

```
%use equality/rationals.
```

Extensions must be installed prior to their use. For this reason, it is a safe practice to put all the `%use` declarations at the beginning of the program.

6.2 Equalities over Rational Numbers

As mentioned before, the extension presiding equality over the rational numbers is called `equality/rationals`, and it is therefore installed by the declaration

```
%use equality/rationals.
```

This causes the declarations

```
rational : type.

~ : rational -> rational.               %prefix ~ 9999.
+ : rational -> rational -> rational.   %infix + left 9997.
- : rational -> rational -> rational.   %infix - left 9997.
* : rational -> rational -> rational.   %infix * left 9998.
```

to be included in the current global signature.

Note that We do not add an equality predicate for rationals. This is unnecessary, since the extension modifies standard type checking so that arithmetic identities are taken into account. However, one can always define an equality predicate by declaring

```
== : rational -> rational -> type.  %infix none 100 ==.
id : X == X.
```

This extension is also responsible for defining special constants for all the rational numbers. These follow the syntax

```
<rational> ::= ~<unsigned> | <unsigned>
<unsigned> ::= <digits> | <digits>/<digits>
<digits>   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
             | <digits> <digits>
```

It is important to notice the difference between `~ 10` and `~10`. The former is the object obtained applying the unary minus operator to the positive number 10; the second stands for the number -10. The difference is, however, just syntactical, since the former is automatically evaluated into the latter by Twelf(X). In general, one should keep in mind that Twelf(X) extensions do not modify the behavior of the lexer; hence, for example, multiplication of a variable `X` by two still needs to be written as `X * 2` (note the spaces preceding and following the multiplication symbol), while `X*2` will be interpreted by the system as a single variable named "`X*2`".

Unification of arithmetic expressions is done by Gaussian elimination. Thus, unification problems that can be reduced to linear equations over existentially quantified variables are immediately solved; other problems not falling in this class are likely to be delayed as unificational constraints.

6.3 Inequalities over Rational Numbers

Arithmetical inequalities are handled by the extension `inequality/rationals`. This relies on `equality/rationals` for the definition of rational number, so a declaration of

```
%use inequality/rationals.
```

implicitly entails one of

```
%use equality/rationals.
```

This extension adds the following to the signature:

```
>  : rational -> rational -> type.  %infix none 0 >.
>= : rational -> rational -> type.  %infix none 0 >=.

+>  : {Z:rational} X > Y -> (X + Z) > (Y + Z).
+>= : {Z:rational} X >= Y -> (X + Z) >= (Y + Z).

>>=  : X > Y -> X >= Y.
0>=0 : 0 >= 0.
```

It also adds countably many proof objects such as

```
45>0 : 45 > 0.
```

witnessing that positive numbers are greater than zero.

Goals of the form

`M > N` or `M >= N`

are evaluated using a modified version of the simplex algorithm, rather than the usual proof-search mechanism. This will incrementally collect inequalities, assigning them incomplete (i.e. partially instantiated) proof objects, until either an inconsistency is discovered (generating failure) or they can be shown to be satisfied a unique solution (causing the proof objects to be finally completed).

6.4 Integer Constraints

The extensions `equality/integers` and `inequality/integers` deal with equalities and inequalities over the ring of integer numbers, respectively. Since these two modules are very similar to their rationals counterparts, we will limit ourselves to outlining the differences.

The signature introduced by `equality/integers` is roughly the same as `equality/rationals`. The only difference lies in the name of the main type:

```
integer : type.

~ : integer -> integer.             %prefix ~ 9999.
+ : integer -> integer -> integer.  %infix + left 9997.
- : integer -> integer -> integer.  %infix - left 9997.
* : integer -> integer -> integer.  %infix * left 9998.
```

Like before, countably many special constants are added to the signature. They follow the syntax

```
<integer> ::= ~<digits> | <digits>
```

The extension `equality/integers` takes advantage of the fact that (unlike the rationals) the integers are not a dense ordering to considerably shorten the set of symbols needed:

```
>= : integer -> integer -> type.  %infix none 0 >=.

+>= : {Z:integer} X >= Y -> (X + Z) >= (Y + Z).
```

It also declares countably many proof objects such as

```
37>=0 : 37 >= 0.
```

for all non-negative integers.

Notice that strict inequality `>` can be easily defined in this case:

```
> : integer -> integer -> type.  %infix none 0 >.

>=> : X >= (Y + 1) -> X > Y.
```

For solving linear inequalities over the integers, we again use the simplex method, but we add special routines that implement branch-and-bound search. Specifically, inequalities are internally interpreted over rationals, and a check is performed regularly (whenever a new inequality is added) to ensure the rational solution space contains integral points.

Note that all known methods for solving systems of integral linear inequations are notoriously inefficient. In the branch-and-bound method we adopted, the number of branches to consider is potentially exponential with respect to the size of the problem. We recommend using the `inequality/rationals` module instead, whenever the situation allows the two to be used interchangeably.

6.5 Equalities over String

A third domain currently supported by the Twelf(X) extensions are strings of (printable) characters. The only operator provided is string concatenation. Installing

```
%use equality/strings.
```

causes the following signature to be loaded:

```
string : type.
++ : string -> string -> string.   %infix right ++ 9999.
```

String constants are sequences of printable characters enclosed by double quotes:

```
"foobar" : string
```

Under these conventions, strings cannot therefore contain the double quote character `"`. Escape sequences, such as `"\n"` have no special meaning; moreover, we do not currently provide input/output primitives.

Finally, it is required that string constants occupy a single line. The declaration

```
mystring : string = "foo
                     bar".
```

is not considered syntactically correct.

6.6 Sample Constraint Programs

Using Twelf(X), we can write a modified version of `append` that takes into account the size of the list involved:

```
%use equality/rationals.

item : type.
list : rational -> type.

a : type.
b : type.
...

nil  : list 0.
cons : item -> list N -> list (N + 1).

append : list M -> list N -> list (M + N).

append_nil  : append nil L L.
append_cons : append (cons X L1) L2 (cons X L3)
                <- append L1 L2 L3.
```

Type checking the definition of `append` requires some algebraic manipulation. For example, validity of `append_nil` depends on the identity $0+M = M$.

Most classic Constraint Logic Programming (CLP) examples found in the literature can be easily translated to Twelf(X). We present here a simple mortgage calculator:

```
%use inequality/rationals.

%% equality
== : rational -> rational -> type.  %infix none 1000 ==.
id : X == X.

%% mortgage payments
mortgage : rational -> rational -> rational
                    -> rational -> rational -> type.

m0 : mortgage P T I MP B
       <- T > 0
       <- 1 >= T
       <- Interest == T * P * I * 1/1200
       <- B == P + Interest - (T * MP).
m1 : mortgage P T I MP B
       <- T > 1
       <- MI == P * I * 1/1200
       <- mortgage (P + MI - MP) (T - 1) I MP B.
```

This CLP program takes four parameters: the principal `P`, the life of the mortgage in months `T`, the annual interest rate (%) `I`, the monthly payment `MP`, and the outstanding balance `B`.

Finally, we use the string extensions to write a simple parser. Consider the following syntax for simple arithmetic expressions:

```
<expr> ::= <number> | <expr>+<expr>
                    | <expr>-<expr>
                    | <expr>*<expr>
                    | (<expr>)
<number>     ::= <digit> | <digit><number>
<digit>      ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

This can be translated quite directly into Twelf by constructing parsing predicates `digit`, `number`, `expression` as follows:

```
digit : string -> type.

d0 : digit "0".
d1 : digit "1".
d2 : digit "2".
d3 : digit "3".
d4 : digit "4".
d5 : digit "5".
d6 : digit "6".
d7 : digit "7".
d8 : digit "8".
d9 : digit "9".

number : string -> type.

nd  : number X
        <- digit X.
n++ : number (X ++ Y)
        <- digit X
        <- number Y.

expression : string -> type.

en : expression X
       <- number X.
e* : expression (X ++ "*" ++ Y)
       <- expression X
       <- expression Y.
e+ : expression (X ++ "+" ++ Y)
       <- expression X
       <- expression Y.
e- : expression (X ++ "-" ++ Y)
       <- expression X
       <- expression Y.
```

```
ep : expression ("(" ++ X ++ ")")
       <- expression X.
```

6.7 Restrictions and Caveats

To ensure the consistency of the calculus, use of types defined by Twelf(X) extensions is restricted. Specifically, we do not allow them to be used as dynamic assumptions. This rules out meaningless (under the current operating semantics) goals like

```
?- 0 > 1 -> foo.
```

as well as meaningful, but intractable ones, such as

```
?- {X : rational} X * X >= 0
```

One important thing to keep in mind when using arithmetic is that this currently is defined over the rationals, rather than the integers or the set of natural numbers. While in many applications this is of no consequence, it might generate in some cases surprising results. While it is certainly possible to define the characteristic function of the integer subset (see the content of `clp-examples/integers/` in the distribution) and use this to enforce all computations to take place in the intended domains, this introduces a big performance penalty and should in general be avoided.

7 Modes

In most cases, the correctness of the algorithmic interpretation of a signature as a logic program depends on a restriction to queries of a certain form. Often, this is a restriction of some arguments to *inputs* which must be given as *ground* objects, that is, objects not containing any existential variables. In return, one often obtains *outputs* which will also be ground. In the logic programming terminology, the information about which arguments to a predicate should be considered input and output is called *mode information*.

Twelf supports a simple system of modes. It checks explicit mode declarations by the programmer against the signature and signals errors if the prescribed information flow is violated. Currently, queries are not checked against the mode declaration.

Mode checking is useful to uncover certain types of errors which elude the type-checker. It can also be used to generate more efficient code, although the compiler currently does not take advantage of mode information.

There are two forms of mode declarations: a short form which is adequate and appropriate most of the time, and a long form which is sometimes necessary to ascribe the right modes to implicit arguments

```
mdecl ::= smdecl     % short mode declaration
        | fmdecl     % full mode declaration

decl ::= ...
       | %mode mdecl.
```

7.1 Short Mode Declaration

There are two forms of mode declarations: a short and a full form. The short form is an abbreviation which is expanded into the full form when it is unambiguous.

```
mode ::= +          % input
       | *          % unrestricted
       | -          % output

mid ::= mode id  % named mode identifier, one token

smdecl ::= id           % type family a
         | smdecl mid  % argument mode
```

Mode declarations for a predicate `a` must come before any clauses defining `a`. Note that the mode followed with the identifier must be one token, such as '`+L`' and not '`+ L`'. The short form is most convenient in typical situations. For example, we can declare that the `append` program (see Section 5.4 [Sample Trace], page 21) takes the first two arguments as input and produces the the third as output.

```
append  : list -> list -> list -> type.
%mode append +L +K -M.
```

If we wanted to use `append` to split a list into two sublists, we would instead declare

```
append : list -> list -> list -> type.
%mode append -L -K +M.
```

where the clauses `appNil` and `appCons` remain unchanged.

In the lambda-calculus type checker (see Section 5.6 [Sample Program], page 24), the type must be an unrestricted argument.

```
of : exp -> tp -> type.
%mode of +E *T.
```

If we declare it as an input argument, `%mode of +E +T`, we obtain an error pointing to the first occurrence of `T2` in the clause `tp_app` reproduced below.

```
examples/nd/lam.elf:27.20-27.22 Error:
Occurrence of variable T2 in input (+) argument not necessarily ground

tp_app : of (app E1 E2) T1
          <- of E1 (arrow T2 T1)
          <- of E2 T2.
```

If we declare it as an output argument, `%mode of +E -T`, we obtain an error pointing to the second occurrence of `T1` in the clause `tp_lam` reproduced below.

```
examples/nd/lam.elf:25.8-25.10 Error:
Occurrence of variable T1 in output (-) argument not necessarily ground

tp_lam : of (lam E) (arrow T1 T2)
          <- ({x:exp}
                of x T1 -> of (E x) T2).
```

In general, for a mode declaration in short form the arguments are specified exactly as they would look in the program. This means one cannot specify the modes of implicit arguments which are filled in by term reconstruction. These modes are reconstructed as follows: each implicit argument which appears in the type of an input argument is considered input '`+`', those among the remaining which appear in an output argument are considered output '`-`', the rest are unrestricted. The mode declaration is echoed in full form, so the user can verify the correctness of the modes assigned to implicit arguments. If the inferred full mode declaration is incorrect, or if one wants to be explicit about modes, one should use full mode declarations (see Section 7.2 [Full Mode Declaration], page 39).

7.2 Full Mode Declaration

To specify modes for implicit arguments one must use the full form of mode declaration. A `mode` can be one of '`+`', '`*`', or '`-`' (see Section 7.1 [Short Mode Declaration], page 37).

```
fmdecl ::= mode {id : term} fmdecl
         | mode {id} fmdecl
         | term
```

The term following the mode prefix in a full mode declaration must always have the form $a\ x1 \ldots xn$ where $x1$ through xn are variables declared in the mode prefix. As an example, we give an alternative specification of the `append` predicate.

```
append : list -> list -> list -> type.
%mode +{L:list} +{K:list} -{M:list} append L K M.
```

7.3 Mode Checking

Mode checking for input, output, and unrestricted arguments examines each clause as it is encountered. The algorithm performs a kind of abstract interpretation of the clause, keeping track of a list of the existential variables for which it knows that they will be ground.

1. We assume each existential variable with a strict occurrence (see Section 4.3 [Strict Occurrences], page 15) in an input argument to the clause head to be ground.
2. We traverse the subgoals in evaluation order (see Section 5.5 [Operational Semantics], page 23). For each subgoal we first verify that all input arguments will be ground, using the information about the existential variables collected so far. If this check succeeds we add all variables which have a strict occurrence in an output argument of the subgoal to the list of variables with known ground instantiations.
3. After the last subgoal has been examined, we verify that the output arguments in the clause head are now also ground.

Arguments whose mode is unrestricted are ignored: they do no need to be checked, and they do not contribute any information about the instantiations of existential variables.

8 Termination

Besides checking types and modes, Twelf can also verify if a given type family, when interpreted as a logic program, always terminates on well-moded goals. In many cases this means that the program implements a decision procedure. Of course, in general termination is undecidable, so we only check a simple sufficient condition.

Checking termination presupposes that the program is well-typed and guarantees termination only when the arguments involved in the termination order are ground. This will always be true for well-moded goals, since mode and termination declarations must be consistent.

Termination is different from checking types and modes in that it is not checked incrementally as the signature is read. Instead, termination of a predicate is a global property of the program once it has been read. Thus termination declarations came after the predicate has been fully defined; further extensions of the predicate are not checked and may invalidate termination.

The termination checker is rather rudimentary in that it only allows lexicographic and simultaneous extensions of the subterm ordering. Moreover, it does not take into account if a result returned by a predicate is smaller than an input argument. Nonetheless, for the style of programs written in Twelf, the termination of many decision procedures can be verified.

8.1 Termination Declaration

The termination orders we construct are lexicographic or simultaneous extensions of the subterm ordering explained in Section 8.3 [Subterm Ordering], page 45. The termination declaration associates the termination order with argument positions of predicates via call patterns.

The case of mutually recursive predicates is particularly complex and requires mutual call patterns and mutual arguments. Their syntax is given below; they are explained in Section 8.6 [Mutual Recursion], page 48.

```
args ::=
       | id args        % named argument
       | _ args         % anonymous argument

callpat ::= id args     % a x1 ... xn

callpats ::=
           | (callpat) callpats
                        % mutual call patterns
ids ::=
      | id ids          % argument name

marg ::= id             % single argument
       | ( ids )        % mutual arguments

orders ::=
         | order orders % component order

order ::= marg          % subterm order
        | { orders }    % lexicographic order
        | [ orders ]    % simultaneous order

tdecl ::= order callpats    % termination order

decl ::= ...
       | %terminates tdecl. % termination declaration
```

All identifiers in the order specification of a termination declaration must be upper case, must occur in the call patterns, and no variable may be repeated. Furthermore, all arguments participating in the termination order must occur in the call patterns in input positions.

The most frequent form of termination declaration is

```
%terminates Xi (a X1 ... Xn).
```

which expresses that predicate `a` terminates because recursive calls decrease the input argument `Xi` according to the subterm ordering (see Section 8.3 [Subterm Ordering], page 45).

As an example, we consider a proof that simple type inference (see Section 5.6 [Sample Program], page 24) terminates. Recall the relevant program fragment (see '`examples/guide/lam.elf`').

```
of : exp -> tp -> type.                %name of P.
%mode of +E *T.

tp_lam : of (lam E) (arrow T1 T2)      % |- lam x. E : T1 => T2
          <- ({x:exp}                  % if  x:T1 |- E : T2.
                of x T1 -> of (E x) T2).

tp_app : of (app E1 E2) T1             % |- E1 E2 : T1
          <- of E1 (arrow T2 T1)       % if  |- E1 : T2 => T1
          <- of E2 T2.                 % and |- E2 : T2.
```

The typability of an expression is always reduced to the typability of its subexpressions. Therefore any call to the `of` predicate with a ground expression should terminate. In general, termination can only be checked for input arguments, and all calls must be well-moded (see Section 7.3 [Mode Checking], page 40). Twelf verifies termination with the declaration

```
%terminates E (of E T).
```

Here, `E` specifies the decreasing argument, namely the first argument of the typing judgment as expressed in the call pattern `(of E T)`.

A corresponding attempt to show that evaluation always terminates,

```
%terminates E (eval E V).
```

fails for the clause `ev_app` with the message

```
examples/guide/lam.elf:1053-1068 Error:
Termination violation:
(E1' V2) < (app E1 E2)
```

indicating that in a recursive call the term `E1' V2` could not be shown to be smaller than `app E1 E2`. In our example, of course, evaluation need not terminate for precisely this reason.

Termination checking plays a crucial role in checking meta-programs. The meta-program represents a proof that some relation about programs holds. The recursive calls in the meta-program correspond to applications of the induction hypothesis in the proof. Termination checking of meta-programs corresponds to checking that the application of the induction hypothesis is valid, i.e. we apply the induction hypothesis to a smaller argument.

8.2 Reduction Declaration

A reduction predicate specifies a relation between input and output arguments of a program. The reduction checker is rather restrictive and allows only relations based on subterm ordering. For example, we cannot reason about the length of a list or term. Moreover, reduction checking presupposes that the right side of the reduction predicate corresponds to the input arguments and the left side of the predicate corresponds to the output arguments. The declaration `%reduces` checks if the specified reduction predicate holds for a given program. To check whether the predicate also terminates, one needs to check termination of the predicate separately. The syntax for `order` can be found in Chapter 8 [Termination], page 41.

```
pdecl ::= order < order      % strictly smaller than
        | order <= order     % smaller or equal than
        | order = order      % equal

rdecl ::= pdecl callpats     % reduction predicate

decl ::= ...
       | %reduces rdecl.     % reduction declaration
```

A `pdecl` requires that both orders specified are of the same variety. For example, if one is lexicographic, the other one must be as well.

For example,

```
%reduces Z <= X (minus X Y Z).
```

expects `X` to be the input and `Z` to be the output of the goal `(minus X Y Z)`. The declaration checks whether the output argument `Z` is smaller than the input argument `X`, and if the program terminates in `X`.

As a simple example of reduction checking we consider the greatest common divisor. This is an exceprt from the signature for unary arithmetic in the file '`example/guide/arith.elf`'.

```
gcd: nat -> nat -> nat -> type.          %name gcd G.
%mode gcd +X +Y -Z.

gcd_z1: gcd z Y Y.
gcd_z2: gcd X z X.

gcd_s1: gcd (s X) (s Y) Z
  <- less (s X) (s Y) true
  <- rminus (s Y) (s X) Y'
  <- gcd (s X) Y' Z.

gcd_s1: gcd (s X) (s Y) Z
  <- less  (s X)(s Y) false
  <- rminus (s X) (s Y) X'
  <- gcd X' (s Y)  Z.

rminus: nat -> nat -> nat -> type.        %name rminus M.
%mode rminus +X +Y -Z.
rmin : rminus (s X) (s Y) Z
 <- sub X Y Z.

%reduces Z < X (rminus X Y Z).
%terminates [X Y] (gcd X Y _).
```

In order to verify that the definition of gcd terminates, we need to show that the arguments in the recursive call are decreasing, i.e. we need to show `Y' < (s Y)` and `X' < (s X)`. To check that these properties hold, we need the fact that the output argument `Y'` of `rminus (s Y) (s X) Y'` is always smaller than the input argument `(s Y)` (i.e. `rminus (s Y) (s X) Y'` always satisfies the reduction predicate `Y' < (s Y)`). We verify that `rminus (s Y) (s X) Y'` always reduces its output argument by checking the declaration `%reduces Z < X (rminus X Y Z)`. While checking termination of `gcd` we use this information and prove `Y' < (s Y)` (termination condition of `gcd`) under the assumption `Y' < (s Y)` (reduction predicate of `rminus`).

8.3 Subterm Ordering

On first-order terms, that is, terms not containing lambda-abstraction, the subterm ordering is familiar: $M<N$ if M is a strict subterm of N, that is, M is a subterm N and M is different from N.

On higher-order terms, the relation is slightly more complicated because we must allow the substitution of parameters for bound variables without destroying the subterm relation. Consider, for example, the case of the typing rule

```
of : exp -> tp -> type.                %name of P.
%mode of +E *T.

tp_lam : of (lam E) (arrow T1 T2)      % |- lam x. E : T1 => T2
          <- ({x:exp}                  % if  x:T1 |- E : T2.
                of x T1 -> of (E x) T2).
```

from the signature for type inference (see Section 5.6 [Sample Program], page 24) in the file '`example/guide/lam.elf`'. We must recognize that

```
(E x) < (lam E)
```

according to the subterm ordering. This is because `E` stands for a term `[y:exp] E'` and so `E x` has the same structure as `E'` except that `y` (a bound variable) has been replaced by `x` (a parameter). This kind of pattern arises frequently in Twelf programs.

On the other hand, the restriction to parameter arguments of functions is critical. For example, the lax rule

```
tp_applam : of (app (lam E1) E2) T2
             <- of (E1 E2) T2.
```

which applies `E1` to `E2` which is not a parameter, is indeed not terminating. This can be seen from the query

```
?- of (app (lam [x:exp] app x x) (lam [y:exp] app y y)) T.
```

The restriction of the arguments to parameters can be lifted when the type of the argument is not mutually recursive with the result type of the function. For example, the signature for natural deduction (see Section 3.6 [Sample Signature], page 11, contains no constructor which allows propositions to occur inside individual terms. Therefore

```
(A T) < (forall A)
```

where `A : i -> o` and `T : i` is an arbitrary term (not just a parameter). Intuitively, this is correct because the number of quantifiers and logical connectives is smaller on the left, since `T` cannot contain such quantifiers or connectives.

This kind of precise analysis is important, for example, in the proof of cut elimination or the termination of polymorphic type reconstruction.

8.4 Lexicographic Orders

Lexicographic orders are specified as

{*O1* ... *On*}

Using vi and wi for corresponding argument structures whose order is already defined, we compare them lexicographically as follows:

$\{v1 \ldots vn\} < \{w1 \ldots wn\}$, if

$v1 {<} w1$, or

$v1 = w1$ and $v2 {<} w2$, or

...

$v1 = w1$, $v2 = w2$, ..., and $vn {<} wn$.

A lexicographic order is needed, for example, to show termination of Ackermann's function, defined in '`examples/arith/arith.elf`' with the termination declaration in '`examples/arith/arith.thm`'.

8.5 Simultaneous Orders

Simultaneous orders require that one of its elements decreases while all others remain the same. This is strictly weaker than a lexicographic ordering built from the same components. Technically speaking it is therefore is redundant for termination checking, since the corresponding lexicographic ordering could be used instead. However, for inductive theorem proving it is quite useful, since the search space for simultaneous induction is much smaller than for lexicographic induction.

Simultaneous orders are specified as

[*O1* ... *On*]

Using vi and wi for corresponding argument structures whose order is already defined, we compare them simultaneously as follows:

$[v1 \ldots vn] < [w1 \ldots wn]$, if

$v1 {<} w1$, $v2 {<} = w2$, ..., and $vn {<} = wn$, or

$v1 <= w1$, $v2 < w2$, ..., and $vn <= wn$, or

...

$v1 <= w1$, $v2 <= w2$, ..., and $vn < wn$.

A combination of simultaneous and lexicographic order is used, for example, in the admissibility of cut found in '`examples/cut-elim/int.thm`', where either the cut formula A gets smaller, or if A stays the same, either the derivation of the left or right premise get smaller while the other stays the same.

8.6 Mutual Recursion

Mutually recursive predicates present a challenge to termination checking, since decreasing arguments might appear in different positions. Moreover, mutually recursive predicates `a` and `a'` might be prioritized so that when `a` calls `a'` all termination arguments remain the same, but when `a'` calls `a` the arguments are smaller according to the termination order.

To handle the association of related argument in mutually recursive predicates, so-called *mutual arguments* can be specified in a termination order. They are given as

```
(X1 ... Xn)
```

The priority between predicates is indicated by the order of the call patterns. If we analyze call patterns

```
(a1 args1)
(a2 args2)
...
(an argsn)
```

then `ai` may call `aj` for $i<j$ with equal termination arguments, but calls of `ai` from `aj` must decrease the termination order.

Mutual arguments are used, for example, in the proofs of soundness (file '`examples/lp-horn/uni-sound.thm`') and completeness (file '`examples/lp-horn/uni-complete.thm`') of uniform derivations for Horn logic.

9 Theorem Prover

Disclaimer: The theorem proving component of Twelf is in an even more experimental stage and currently under active development.

Nonetheless, it can prove a number of interesting examples automatically which illustrate our approach the meta-theorem proving which is described in *Schuermann and Pfenning 1998, CADE*. These examples include type preservation for Mini-ML, one direction of compiler correctness for different abstract machines, soundness and completeness for logic programming interpreters, and the deduction theorem for Hilbert's formulation of propositional logic. These and other examples can be found in the example directories of the Twelf distribution (see Chapter 14 [Examples], page 83).

A *theorem* in Twelf is, properly speaking, a meta-theorem: it expresses a property of objects constructed over a fixed LF signature. Theorems are stated in the meta-logic M2 whose quantifiers range over LF objects. In the simplest case, we may just be asserting the existence of an LF object of a given type. This only requires direct search for a proof term, using methods inspired by logic programming. More generally, we may claim and prove forall/exists statements which allow us to express meta-theorems which require structural induction, such as type preservation under evaluation in a simple functional language (see Section 5.6 [Sample Program], page 24).

9.1 Theorem Declaration

The theorem proving component of Twelf is in an experimental stage and currently under active development. This documentation describes the present intermediate state.

There are three forms of declarations related to the proving of theorems and meta-theorems. The first, `%theorem`, states a theorem as a meta-formula (`mform`) in the meta-logic M2 defined below. The second, `%prove`, gives a resource bound, a theorem, and an induction ordering and asks Twelf to search for a proof. After a `%prove` declaration succeeds, the theorem will be made available as a lemma to subsequent proofs. In order to avoid that, Twelf offers the form `%establish` which is like `%prove`, but the established theorem will never be used in subsequent proofs.

Note that a well-typed `%theorem` declaration always succeeds, while the `%prove` and `%establish` declarations only succeed if Twelf can find a proof.

```
dec ::= {id:term}          % x : A
      | {id}               % x

decs ::= dec
       | dec decs

ctx ::= some decs pi decs
      | some decs pi decs | ctx

mform ::= forallG ctx mform  % regular contexts
        | forall* decs mform % implicit universal
        | forall decs mform  % universal
        | exists decs mform  % existential
        | true               % truth

thdecl ::= id : mform        % theorem name a, spec

pdecl ::= nat order callpats % bound, induction order, theorems

decl ::= ...
       | %theorem thdecl.   % theorem declaration
       | %prove pdecl.      % prove declaration
       | %establish pdecl.  % prove declaration, do not use as lemma later
       | %assert callpats.  % assert theorem (requires Twelf.unsafe)
```

The prover only accepts quantifier alternations of the form `forall*` *decs* `forall` *decs* `exists` *decs* `true`. Note that the implicit quantifiers (which will be suppressed when printing the proof terms) must all be collected in front, but after the specification of the regular contexts.

The syntax and meaning of `order` and `callpats` are explained in Chapter 8 [Termination], page 41, since they are also critical notions in the simpler termination checker.

9.2 Sample Theorems

As a first example, we use the theorem prover to establish a simple theorem in first-order logic (namely that A implies A for any proposition A), using the signature for natural deduction (see Section 3.6 [Sample Signature], page 11).

```
%theorem
trivI : exists {D:{A:o} nd (A imp A)}
        true.

%prove 2 {} (trivI D).
```

The empty termination ordering {} instructs Twelf not to use induction to prove the theorem. The declarations above succeed, and with the default setting of 3 for Twelf.chatter we see

```
%theorem trivI : ({A:o} nd (A imp A)) -> type.
%prove 2 {} (trivI D).
%QED
%skolem trivI#2 : {A:o} nd (A imp A).
```

The line starting with %theorem shows the way the theorem will be realized as a logic program predicate. In earlier versions this was such a logic program was actually constructed; at present this feature has been disabled while the implementation has been improved to allow regular contexts.

The second example is the type preservation theorem for evaluation in the lambda-calculus. This is a continuation of the example in Section Section 5.6 [Sample Program], page 24 in the file 'examples/guide/lam.elf'. Type preservation states that if and expression E has type T and E evaluates to V, the V also has type T. This is expressed as the following %theorem declaration.

```
%theorem
tps : forall* {E:exp} {V:exp} {T:tp}
       forall {D:eval E V} {P:of E T}
       exists {Q:of V T}
       true.
```

The proof proceeds by structural induction on D, the evaluation from E to V. Therefore we can search for the proof with the following declaration (where the size bound of 5 on proof term size is somewhat arbitrary).

```
%prove 5 D (tps D P Q).
```

Twelf finds and displays the proof easily. The resulting program is installed in the global signature and can then be used to apply type preservation in subsequent proofs (see Section 9.5 [Proof Realizations], page 54).

The third example illustrates the use of regular contexts. We use the theorem prover to establish a simple theorem: If we have to different but structurally identical formulations of the lambda-calculus (exp, exp'), and a relation that shows how we can copy from one to the other, than this relation is total. In words, this theorem reads as: If E is an expression then there exists an expression E', s.t. E' is a structurally identical copy to E.

```
exp : type.

app : exp -> exp -> exp.
lam : (exp -> exp) -> exp.

exp' : type.

app' : exp' -> exp' -> exp'.
lam' : (exp' -> exp') -> exp'.

cp : exp -> exp' -> type.
cp_app : cp (app D1 D2) (app' E1 E2)
         <- cp D1 E1
         <- cp D2 E2.
cp_lam : cp (lam ([x:exp] D x)) (lam' ([y:exp'] E y))
         <- ({x:exp} {y:exp'} cp x y -> cp (D x) (E y)).


%theorem cpt :  forallG (pi {x:exp} {y:exp'} {u:cp x y})
                forall {E:exp}
                exists {E':exp'} {D: cp E E'}
                true.
%prove 5 {E} (cpt E _ _).
```

It is easy to see, that the `E` and the `E'` cannot be assumed to be closed, because otherwise the induction hypothesis cannot be applied. In this situation, the induction hypothesis must be applied to an object under some additional assumptions, which are `x:exp, y:exp', u:cp x y`, and which we call a context block. Clearly, one context block is not enough; in general `E` can occur in any context regularly built out of these blocks. The `forallG` allows an inductive definition of a *regular context*, abstractly describing the form of actual contexts. Each context block consists of a `some` part which declares additional variables which are instantiated with some well-typed object in a real instance of a context, and a `pi` part which describes the parameters.

The termination ordering `{E}` instructs Twelf to do induction over `E` to prove the theorem. The `%prove` command executes the proof search. In addition, if a proof has been found, the lemma is made accessible to the proof search evoked by subsequent theorems and lemmas, and which might slow it down accordingly. If a lemma is not used in subsequent proofs, the user can use `%establish` instead of `%prove` and it will not be made available.

For certain theorems, the theorem prover will not be able to find a proof, even that it should. This behavior could be caused by an incompleteness in the implementation (which still exist, but which should be removed in the next release of Twelf), or a enormously huge search space, which disallows the underlying LF theorem to construct a proof term. In these situations, one can still try to prove subsequent theorem and lemmas by asserting the correctness of the lemma in question. This is done by the `%assert` command. For the theorem above, one could

```
%assert (cpt _ _ _).
```

after the `Twelf.unsafe` mode has been activated.

9.3 Proof Steps

We expect the proof search component of Twelf to undergo major changes in the near future, so we only briefly review the current state.

Proving proceeds using three main kinds of steps:

Filling
: Using iterative deepening, Twelf searches directly for objects to fill the existential quantifiers, given all the constants in the signature and the universally quantified variables in the theorem. The number of constructors in the answer substitution for each existential quantifier is bounded by the size which is given as part of the `%prove` declaration, thus guaranteeing termination (in principle).

Recursion
: Based on the termination ordering, Twelf appeals to the induction hypothesis on smaller arguments. If there are several ways to use the induction hypothesis, Twelf non-deterministically picks one which has not yet been used. Since there may be infinitely many different ways to apply the induction hypothesis, the parameter `Twelf.Prover.maxRecurse` bounds the number of recursion steps in each case of the proof.

Splitting
: Based on the types of the universally quantified variables, Twelf distinguishes all possible cases by considering all constructors in the signatures. It nevers splits a variable which appears as an index in an input argument, and if there are several possibilities it picks the one with fewest resulting cases. Splitting can go on indefinitely, so the paramater `Twelf.Prover.maxSplit` bounds the number of times a variable may be split.

9.4 Search Strategies

The basic proof steps of filling, recursion, and splitting are sequentialized in a simple strategy which never backtracks. First we attempt to fill all existential variables simultaneously. If that fails we recurse by trying to find new ways to appeal to the induction hypothesis. If this is not possible, we pick a variable to distinguish cases and then prove each subgoal in turn. If none of the steps are possible we fail.

This behavior can be changed with the parameter `Twelf.Prover.strategy` which defaults to `Twelf.Prover.FRS` (which means Filling-Recursion-Splitting). When set to `Twelf.Prover.RFS` Twelf will first try recursion, then filling, followed by splitting. This is often faster, but fails in some cases where the default strategy succeeds.

9.5 Proof Realizations

Proofs of meta-theorems can be realized as logic programs. This is presently disabled. We still describe the possibility below in anticipation of future versions. On the other hand, theorems that have been proved will be skolemized and used in proof of subsequent theorems. However, they will not be used for search.

A logic program is a relational representation of the constructive proof and can be executed to generate witness terms for the existentials from given instances of the universal quantifiers. As an example, we consider once more type preservation (see Section 9.2 [Sample Theorems], page 50).

After the declarations,

```
%theorem
tps : forall* {E:exp} {V:exp} {T:tp}
       forall {D:eval E V} {P:of E T}
       exists {Q:of V T}
       true.

%prove 5 D (tps D P Q).
```

Twelf answers

```
/tps/tp_lam/ev_lam/:
   tps ev_lam (tp_lam ([x:exp] [P2:of x T1] P1 x P2))
      (tp_lam ([x:exp] [P3:of x T1] P1 x P3)).

/tps/tp_app/ev_app/tp_lam/:
   tps (ev_app D1 D2 D3) (tp_app P1 P2) P6
      <- tps D3 P2 (tp_lam ([x:exp] [P4:of x T2] P3 x P4))
      <- tps D2 P1 P5
      <- tps D1 (P3 E5 P5) P6.
```

which is the proof of type preservation expressed as a logic program with two clauses: one for evaluation of a lambda-abstraction, and one for application. Using the `%solve` declaration (see Section 5.2 [Solve Declaration], page 20) we can, for example, evaluate and type-check the identity applied to itself and then use type preservation to obtain a typing derivation for the resulting value.

```
e0 = (app (lam [x] x) (lam [y] y)).
%solve p0 : of e0 T.
%solve d0 : eval e0 V.
%solve tps0 : tps d0 p0 Q.
```

Recall that `%solve c : V` executes the query `V` and defines the constant `c` to abbreviate the resulting proof term.

10 ML Interface

The Twelf implementation defines a number of ML functions embedded in structures which can be called to load files, execute queries, and set environment parameters such as the verbosity level of the interaction. These functions and parameters are available in the `Twelf` structure. If you open the `Twelf` structure with

```
open Twelf
```

after compiling and loading Twelf, you do not have to type the '`Twelf.`' to the functions shown below.

Previous implementations of Elf offered a stand-alone command interpreter but this has not yet been ported. To exit Twelf and ML call `Twelf.OS.exit ();`.

10.1 Configurations

Groups of Twelf files are managed in *configurations*. A configuration is defined by a file, by convention called '`sources.cfg`', which resides in the same directory as the Twelf source files. The configuration file must contain at most one Twelf source file per line, and the files must be listed in dependency order. A configuration *config* can then be defined from the file by the ML declaration

```
val config = Twelf.Config.read "sources.cfg";
```

By convention, the filenames end in the extensions

'`.elf`' for constant declarations and definitions or mixed files,

'`.quy`' for files which contain query declarations,

'`.thm`' for files which contain `%theorem` and `%proof` declarations.

File names may not contain whitespace. They are interpreted relative to the current working directory of ML, but resolved into absolute path names when the configuration file is read. To change the current working directory call

```
Twelf.OS.getDir ();               (* get working directory *)
Twelf.OS.chDir "directory"; (* change working directory *)
```

As an example, we show how the Mini-ML configuration is defined and loaded, assuming your current working directory is the root directory of Twelf.

```
Twelf.make "examples/mini-ml/sources.cfg";
```

The call to `Twelf.make` returns either `Twelf.OK` or `Twelf.ABORT`. It reads each file in turn, starting from an empty signature, printing the results of type reconstruction and search based on the value of the `Twelf.chatter` variable (see Section 10.3 [Environment Parameters], page 59). If another configuration or file has previously been read, all the declarations will first be deleted so that `Twelf.make` always starts from the same state.

Loading a configuration will stop at the first error encountered, issue an appropriate message and return `Twelf.ABORT`. If there is an unexpected internal error (which indicates a bug in the Twelf implementation), it raises an uncaught exception instead and returns to the ML top-level.

To explore the behavior of programs interactively, you may call the Twelf top-level with

```
Twelf.top ();
```

which is explained in Section 5.3 [Interactive Queries], page 20.

10.2 Loading Files

Twelf also allows direct management of the signature by loading individual files. This is generally not recommended because successive declarations simply accumulate in the global signature which may lead to unexpected behavior. The relevant function calls are

```
Twelf.reset ();
Twelf.loadFile "file";
```

where `Twelf.reset ()` resets the current global signature to be empty and `Twelf.readFile "`*file*`"` loads the given *file* whose name is interpreted relative to the current working directory.

Caution: Reading a file twice will not replace the declarations of the first pass by the second, but simply add them to the current signature. If names are reused, old declarations will be shadowed, but they are still in the global signature and might be used in the search for a solution to a query or in theorem proving, leading to unexpected behavior. When in doubt, use configurations (see Section 10.1 [Configurations], page 57) or call `Twelf.reset ()`.

10.3 Environment Parameters

Various flags and parameters can be used to modify the behavior of Twelf and the messages it issues. They are given below with the assignment of the default value.

`Twelf.chatter := 3;`
: Controls the detail of the information which is printed when signatures are read.

  `0` Nothing.

  `1` Just file names.

  `2` File names and number of query solutions.

  `3` Each declarations after type reconstruction.

  `4` Debug information.

  `5` More debug information.

`Twelf.doubleCheck := false;`
: If `true`, each declaration is checked again for type correctness after type reconstruction. This is expensive and useful only for your peace of mind, since type checking is significantly simpler than type reconstruction.

`Twelf.unsafe := false;`
: If `true` it will allow the `%assert` declaration to assert theorems without proof.

`Twelf.Print.implicit := false;`
: If `true`, implicit arguments (normally elided) are printed. Sometimes this is useful to track particularly baffling errors.

`Twelf.Print.depth := NONE;`
: If `SOME(d)` then terms deeper than level `d` are printed as '`%%`'.

`Twelf.Print.length := NONE;`
: If `SOME(l)` then argument lists longer than `l` are truncated with '`...`'.

`Twelf.Print.indent := 3;`
: Controls the amount of indentation for printing nested terms.

`Twelf.Print.width := 80;`
: The value used to decide when to break lines during printing of terms.

`Twelf.Trace.detail := 1;`
: Controls the detail in information during tracing. See Section 10.5 [Tracing and Breakpoints], page 60

`Twelf.Prover.strategy := Twelf.Prover.FRS;`
: Determines the strategy, where `F`=Filling, `R`=Recursion, and `S`=Splitting. Can also be `Twelf.Prover.RFS`.

`Twelf.Prover.maxSplit := 2;`
: The maximal number of generations of a variable introduced by splitting. Setting is to `0` will prohibit proof by cases.

`Twelf.Prover.maxRecurse := 10;`
: The maximal number of appeals to the induction hypothesis in any case during a proof.

10.4 Signature Printing

Twelf provides two ways to print the current global signature.

```
Twelf.Print.sgn ();
Twelf.Print.prog ();
```

The first prints the signature, using only forward arrows `->`, the second will print the signature interpreted as a logic programming using backward arrows `<-`. Depending on your goals, one or the other might be easier to use.

Output can also be generated in TeX format. The necessary library files can be found in the '`tex/`' subdirectory of the distribution. To print the current signature using TeX format, use

```
Twelf.Print.TeX.sgn ();
Twelf.Print.TeX.prog ();
```

with the same interpretation as the plain text printing commands above.

10.5 Tracing and Breakpoints

Twelf no incorporates some rudimentary tracing facilities for the logic programming interpreter of signatures. This is best used within the Emacs server, but it is also available within the ML Interface.

A tracing specification may be associated with constants in a signature.

`Twelf.Trace.None`
: Do not trace.

`Twelf.Trace.Some ["`*c1*`",...,"`*cn*`"]`
: Trace clauses (object-level constants) or predicates (type families) named *c*1 through *cn*.

`Twelf.Trace.All`
: Trace all clauses and predicates.

One can either suspend the execution when a specified clause or predicate is invoked, or simply trace goals

`Twelf.Trace.trace` *spec*

`Twelf.Trace.break` *spec*

When a breakpoint is set, execution will halt and ask for an action from the user. This consists of a (possible empty) line of input followed by `RET`. Current, the following actions are available.

```
<newline> - continue --- execute with current settings
n - next --- take a single step
r - run --- remove all breakpoints and continue
s - skip --- skip until current subgoals succeeds, is retried, or fails
s n - skip to n --- skip until goal (n) is considered
t - trace --- trace all events
u - untrace --- trace no events
d n - detail --- set trace detail to n (0, 1, or 2)
h - hypotheses --- show current hypotheses
g - goal --- show current goal
i - instantiation --- show instantiation of variables in current goal
v X1 ... Xn - variables --- show instantiation of X1 ... Xn
? for help
```

The detail of the trace information can be set with the variable `Trace.detail :=` *n*`;` to one of

`0`
: print no information

`1`
: print standard information

`2`
: print details of unification

It is possible to examine and reset the state of the currently traced predicates with

`Twelf.Trace.show ();`

`Twelf.Trace.reset ();`

10.6 Timing Statistics

Twelf has a few utilities to collect run-time statistics which are useful mainly for the developers. They are collected in the structure `Timers`. Timing information is cumulative in an ML session.

`Twelf.Timers.show ();`
: Show the value of timers and reset them to zero.

`Twelf.Timers.reset ();`
: Simply reset all timers to zero.

`Twelf.Timers.check ();`
: Display the value of timers, but do not reset them.

Caution: Normally, the various times are exclusive, except that the runtime includes the garbage collection time which is shown separately. However, there is a problem the time for printing the answer substitution to a query is charged both to `Printing` and `Solving`.

10.7 Twelf Signature

For reference, here is the ML signature `TWELF` of the `Twelf` structure which defines most functions and flags relevant to loading and executing Twelf programs.

```
signature TWELF =
sig
  structure Print :
  sig
    val implicit : bool ref           (* false, print implicit args *)
    val depth : int option ref        (* NONE, limit print depth *)
    val length : int option ref       (* NONE, limit argument length *)
    val indent : int ref              (* 3, indentation of subterms *)
    val width : int ref               (* 80, line width *)

    val sgn : unit -> unit            (* print signature *)
    val prog : unit -> unit           (* print signature as program *)

    structure TeX :                   (* print in TeX format *)
    sig
      val sgn : unit -> unit          (* print signature *)
      val prog : unit -> unit         (* print signature as program *)
    end
  end
```

```
structure Trace :
sig
  datatype 'a Spec =                  (* trace and breakpoint spec *)
    None                              (* no tracing, default *)
  | Some of 'a list                   (* list of clauses and families *)
  | All                               (* trace all clauses and families *)

  val trace : string Spec -> unit     (* trace clauses and families *)
  val break : string Spec -> unit     (* break at clauses and families *)
  val detail : int ref                (* 0 = none, 1 = default, 2 = unify *)

  val show : unit -> unit             (* show trace, break, and detail *)
  val reset : unit -> unit            (* reset trace, break, and detail *)
end

structure Timers :
sig
  val show : unit -> unit           (* show and reset timers *)
  val reset : unit -> unit          (* reset timers *)
  val check : unit -> unit          (* display, but not no reset *)
end

structure OS :
sig
  val chDir : string -> unit        (* change working directory *)
  val getDir : unit -> string       (* get working directory *)
  val exit : unit -> unit           (* exit Twelf and ML *)
end

structure Prover :
sig
  datatype Strategy = RFS | FRS     (* F=Fill, R=Recurse, S=Split *)
  val strategy : Strategy ref       (* FRS, strategy used for %prove *)
  val maxSplit : int ref            (* 2, bound on splitting  *)
  val maxRecurse : int ref          (* 10, bound on recursion *)
end

val chatter : int ref               (* 3, chatter level *)
val doubleCheck : bool ref          (* false, check after reconstruction *)
val unsafe : bool ref               (* false, allow %assert without proof *)

datatype Status = OK | ABORT        (* return status *)

val reset : unit -> unit            (* reset global signature *)
val loadFile : string -> Status     (* load file *)
val readDecl : unit -> Status       (* read declaration interactively *)
val decl : string -> Status         (* print declaration of constant *)

val top : unit -> unit              (* top-level for interactive queries *)
```

```
  structure Config :
  sig
    type config                      (* configuration *)
    val suffix : string ref          (* suffix of configuration files *)
    val read : string -> config      (* read config file *)
    val load : config -> Status      (* reset and load configuration *)
    val define : string list -> config (* explicitly define configuration *)
  end

  val make : string -> Status        (* read and load configuration *)

  val version : string               (* Twelf version *)
end;  (* signature TWELF *)
```

11 Twelf Server

The Twelf server is a stand-alone command interpreter which provides the functionality of the `Twelf` structure in ML (see Chapter 10 [ML Interface], page 57), but allows no ML definitions. It is significantly smaller than Standard ML and is the recommended way to interact with Twelf except for developers. Its behavior regarding configurations is slightly different in that the server maintains a current configuration, rather than allowing the binding of names to configurations. Configuration are defined with the `Config.read` command which takes a configuration filename as argument.

In Emacs, the Twelf server typically runs in a process buffer called `*twelf-server*`. The user can select this buffer and directly type commands to the Twelf server. This style of interaction is inherited from the comint package for Emacs, but typically one works through advanced commands in Twelf mode (see Section 12.1 [Twelf Mode], page 69).

The Twelf server prompts with `%% OK %%` or `%% ABORT %%` depending on the success of failure of the previous operation. It accepts commands and their arguments on one line, except that additional Twelf declarations which may be required are read separately, following the command line. Reading declarations can be forcibly terminated with the end-of-file token '`%.`'.

11.1 Server Types

The server commands employ arguments of the following types.

`file` The name of a file, relative to the current working directory.

`id` A Twelf identifier

`strategy` Either `FRS` or `RFS` (see Section 9.4 [Search Strategies], page 53)

`bool` Either `true` or `false`

`nat` A natural number (starting at `0`)

`limit` Either `*` (to indicate no limit) or a natural number

11.2 Server Commands

The Twelf server recognized the following commands.

`set` *parameter value*

Set *parameter* to *value*, where *parameter* is on of the following (explained in Section 10.3 [Environment Parameters], page 59).

`chatter` *nat*

`doubleCheck` *bool*

`Twelf.unsafe` *bool*

`Print.implicit` *bool*

`Print.depth` *limit*

`Print.length` *limit*

`Print.indent` *nat*

`Print.width` *nat*

`Trace.detail` *nat*

`Prover.strategy` *strategy*

`Prover.maxSplit` *nat*

`Prover.maxRecurse` *nat*

`get` *parameter*

Print the current value of *parameter* (see table above).

`Trace.trace` *id1 ... idn*

Trace given constants

`Trace.traceAll`

Trace all constants

`Trace.untrace`

Untrace all constants

`Trace.break` *id1 ... idn*

Set breakpoint for given constants

`Trace.breakAll`

Break on all constants

`Trace.unbreak`

Remove all breakpoints

`Trace.show`

Show current trace and breakpoints

`Trace.reset`

Reset all tracing and breaking

`Print.sgn`

Print current signature

`Print.prog`
: Print current signature as program

`Print.TeX.sgn`
: Print current signature in TeX format

`Print.TeX.prog`
: Print current signature in TeX format as program

`Timers.show`
: Print and reset timers.

`Timers.reset`
: Reset timers.

`Timers.check`
: Print, but do not reset timrs.

`OS.chDir` *file*
: Change working directory to *file*.

`OS.getDir`
: Print current working directory.

`OS.exit`
: Exit Twelf server.

`quit`
: Quit Twelf server (same as exit).

`Config.read` *file*
: Read current configuration from *file*.

`Config.load`
: Load current configuration

`make` *file*
: Read and load current configuration from *file*.

`reset`
: Reset global signature.

`loadFile` *file*
: Load Twelf file *file*.

`decl` *id*
: Show constant declaration for *id*.

`top`
: Enter interactive query loop (see Section 5.3 [Interactive Queries], page 20)

12 Emacs Interface

The Twelf mode for Emacs provides some functions and utilities for editing Twelf source and for interacting with an inferior Twelf server process which can load configurations, files, and individual declarations and track the source location of errors. It also provides an interface to the tags package which allows simple editing of groups of files, constant name completion, and locating of constant declarations within the files of a configuration.

Note that in order to use the Emacs interface you need to include the line

```
(load "directory/emacs/twelf-init.el")
```

in your '.emacs' file, where *directory* is the Twelf root directory.

12.1 Twelf Mode

The Twelf mode in Emacs provides support for editing and indentation, syntax highlighting (including colors) (see Section 12.13 [Syntax Highlighting], page 77), and communication commands for interacting with a Twelf server running as an inferior process to Emacs. It defines a menu which is added to the menu bar, usually at the top of each Emacs frame.

Many commands apply to the current declaration, which is the declaration in which we find the Emacs cursor (not the cursor of the window system). If the cursor is between declarations, the declaration after point is considered current. From the point of view of Emacs, single declarations never include consecutive blank lines, which provides some insulation against missing closing delimiters.

Normally, Twelf mode is entered automatically when a Twelf source file is edited (see Section 12.14 [Emacs Initialization], page 77), but it can also be switched on or off directly with `M-x twelf-mode`.

`M-x twelf-mode`
: Toggle Twelf mode, the major mode for editing Twelf code.

12.2 Editing Commands

The editing commands in Twelf mode partially analyse the structure of the text at the cursor position as Twelf code and try to indent accordingly. This is not always perfect.

`TAB`
`M-x twelf-indent-line`
: Indent current line as Twelf code. This recognizes comments, matching delimiters, and standard infix operators.

`DEL`
`M-x backward-delete-char-untabify`
: Delete character backward, changing tabs into spaces.

`M-C-q`
`M-x twelf-indent-decl`
: Indent each line of the current declaration.

`M-x twelf-indent-region`
: Indent each line of the region as Twelf code.

12.3 Type Checking Commands

The Twelf mode provides simple commands which cause the server to load or reload the current configuration, the file edited in the current buffer, or just the declaration at point. Each of these command can be preceded by a prefix argument (for example, `C-u C-c C-c`) which will select the Twelf server buffer after completion of the command. The Twelf server buffer can also be forced to be shown with the `C-c C-u` Emacs command.

`C-c C-c`
`M-x twelf-save-check-config`
: Save its modified buffers and then check the current Twelf configuration. With prefix argument also displays Twelf server buffer. If necessary, this will start up an Twelf server process.

`C-c C-s`
`M-x twelf-save-check-file`
: Save buffer and then check it by giving a command to the Twelf server. In Twelf Config minor mode, it reconfigures the server. With prefix argument also displays Twelf server buffer.

`C-c C-d`
`M-x twelf-check-declaration`
: Send the current declaration to the Twelf server process for checking. With prefix argument also displays Twelf server buffer.

`C-c c`
`M-x twelf-type-const`
: Display the type of the constant before point. Note that the type of the constant will be 'absolute' rather than the type of the particular instance of the constant.

`C-c C-u`
`M-x twelf-server-display`
: Display Twelf server buffer, moving to the end of output. With prefix argument also selects the Twelf server buffer.

12.4 Printing Commands

`M-x twelf-print-signature`
: Prints the current signature in the Twelf server buffer.

`M-x twelf-print-program`
: Prints the current signature as a program in the Twelf server buffer.

`M-x twelf-print-tex-signature`
: Prints the current signature in TeX style. The output appears in the Twelf server buffer.

`M-x twelf-print-tex-program`
: Prints the current signature as a program in TeX style. The output appears in the Twelf server buffer.

12.5 Tracing Commands

The Twelf Emacs mode provides a simple interface to the tracer. While tracing or breakpoints are on, you should be in the Emacs server buffer to type your input directly to the server as described in Section 10.5 [Tracing and Breakpoints], page 60.

`M-x twelf-trace-trace`
: Read list of constants and trace them.

`M-x twelf-trace-trace-all`
: Trace all clauses and families.

`M-x twelf-trace-untrace`
: Untrace all clauses and families.

`M-x twelf-trace-break`
: Read list of constants and set breakpoints.

`M-x twelf-trace-break-all`
: Set breakpoints on all clauses and families.

`M-x twelf-trace-unbreak`
: Remove all breakpoints.

`M-x twelf-trace-show`
: Show tracing and breakpoint information.

12.6 Error Tracking

Error messages by the Twelf server are flagged with the filename and an educated guess as to the source of the error (see Section 4.6 [Error Messages], page 18). These can be interpreted by Emacs to jump directly to the suspected site.

Sometimes, the server buffer and the the server itself believe to have different working directories. In that case, error tracking may not be able to find the file, and an explicit call to `OS.chDir` or `M-x cd` in the server buffer may be required.

`` C-c ` ``
`M-x twelf-next-error`
: Find the next error by parsing the Twelf server or Twelf-SML buffer. Move the error message on the top line of the window; put the cursor at the beginning of the error source. If the error message specifies a range, the mark is placed at the end.

`C-c =`
`M-x twelf-goto-error`
: Go to the error reported on the current line or below. Also updates the error cursor to the current line.

12.7 Server State

The server state consists of the current configuration and a number of parameters described in Chapter 11 [Twelf Server], page 65. The current configuration is often set implicitly, with the `C-c C-c` command in a configuration buffer, but it can also be set explicitly.

`C-c <`
`M-x twelf-set`
: Sets the Twelf parameter PARM to VALUE. When called interactively, prompts for parameter and value, supporting completion.

`C-c >`
`M-x twelf-get`
: Prints the value of the Twelf parameter PARM. When called interactively, promts for parameter, supporting completion.

`C-c C-i`
`M-x twelf-server-interrupt`
: Interrupt the Twelf server process.

`M-x twelf-server`
: Start an Twelf server process in a buffer named *twelf-server*. Any previously existing process is deleted after confirmation. Optional argument PROGRAM defaults to the value of the variable twelf-server-program. This locally re-binds 'twelf-server-timeout' to 15 secs.

`M-x twelf-server-configure`
: Initializes the Twelf server configuration from CONFIG-FILE. A configuration file is a list of relative file names in dependency order. Lines starting with % are treated as comments. Starts a Twelf servers if necessary.

`M-x twelf-reset`
: Reset the global signature of Twelf maintained by the server.

`M-x twelf-server-quit`
: Kill the Twelf server process.

`M-x twelf-server-restart`
: Restarts server and re-initializes configuration. This is primarily useful during debugging of the Twelf server code or if the Twelf server is hopelessly wedged.

`M-x twelf-server-send-command`
: Restarts server and re-initializes configuration. This is primarily useful during debugging of the Twelf server code or if the Twelf server is hopelessly wedged.

12.8 Info File

The content of this file in Info format can be visited directly and does not need to be tied into the Info tree. See the documentation for the Emacs info package for more info

`C-c C-h`
`M-x twelf-info`
: Visit the Twelf User's Guide in info format in Emacs. With a prefix argument it prompts for the info file name, which defaults to the value of the `twelf-info-file` variable.

12.9 Tags Files

Tags files provide a convenient way to group files, such as Twelf configurations. See the documentation for the Emacs etags package for more information.

`M-x twelf-tag`
: Create tags file for current configuration. If the current configuration is sources.cfg, the tags file is TAGS. If current configuration is named FILE.cfg, tags file will be named FILE.tag Errors are displayed in the Twelf server buffer.

`M-.`
`M-x find-tag TAG`
: Selects the buffer that the tag is contained in and puts point at its definition.

`C-x 4 .`
`M-x find-tag-other-window TAG`
: Selects the buffer that TAG is contained in in another window and puts point at its definition.

`C-c q`
`M-x tags-query-replace FROM TO`
: Query-replace-regexp FROM with TO through all files listed in tags table.

`C-c s`
`M-x tags-search REGEXP`
: Search through all files listed in tags table for match for REGEXP.

`M-,`
`M-x tags-loop-continue`
: Continue last C-c s or C-c q command.

12.10 Twelf Timers

The following commands obtain the runtime statistics of the the Twelf server.

`M-x twelf-timers-reset`
: Reset the Twelf timers.

`M-x twelf-timers-show`
: Show and reset the Twelf timers.

`M-x twelf-timers-check`
: Show the Twelf timers without resetting them.

12.11 Twelf-SML Mode

There is some support for interacting with Twelf, even when it is run within ML, rather than as a stand-alone server. You can start an SML in which you intend to run Twelf with `M-x twelf-sml`; the buffer will then be in Twelf-SML mode.

If you intend to send command to a buffer running Twelf in SML (rather than the Twelf server), you can switch to a minor mode 2Twelf-SML with `M-x twelf-to-twelf-sml`.

`M-x twelf-sml`
: Run an inferior Twelf-SML process in a buffer *twelf-sml*. If there is a process already running in *twelf-sml*, just switch to that buffer. With argument, allows you to change the program which defaults to the value of twelf-sml-program. Runs the hooks from twelf-sml-mode-hook (after the comint-mode-hook is run).

`M-x twelf-to-twelf-sml-mode`
: Toggles minor mode for sending queries to Twelf-SML instead of Twelf server.

`C-c C-e`
`M-x twelf-sml-send-query`
: Send the current declaration to the inferior Twelf-SML process as a query. Prefix argument means switch-to-twelf-sml afterwards.

`C-c C-r`
`M-x twelf-sml-send-region`
: Send the current region to the inferior Twelf-SML process. Prefix argument means switch-to-twelf-sml afterwards.

`C-c RETURN`
`M-x twelf-sml-send-newline`
: Send a newline to the inferior Twelf-SML process. If a prefix argument is given, switches to Twelf-SML buffer afterwards.

`C-c ;`
`M-x twelf-sml-send-semicolon`
: Send a semi-colon to the inferior Twelf-SML process. If a prefix argument is given, switched to Twelf-SML buffer afterwards.

`C-c d`
`M-x twelf-sml-cd DIR`
: Make DIR become the Twelf-SML process' buffer's default directory and furthermore issue an appropriate command to the inferior Twelf-SML process.

`M-x twelf-sml-quit`
: Kill the Twelf-SML process.

12.12 Emacs Variables

A number of Emacs variables can be changed to customize the behavior of Twelf mode. The list below is not complete; please refer to the Emacs Lisp sources in `emacs/twelf.el` for additional information.

`twelf-indent`
: Indent for Twelf expressions.

`twelf-server-program`
: Default Twelf server program.

`twelf-info-file`
: Default Twelf info file.

`twelf-mode-hook`
: List of hook functions to run when switching to Twelf mode.

`twelf-server-mode-hook`
: List of hook functions to run when switching to Twelf Server mode.

`twelf-sml-program`
: Default Twelf-SML program.

`twelf-sml-mode-hook`
: List of hook functions for Twelf-SML mode.

12.13 Syntax Highlighting

Twelf also provides syntax highlighting, which helps make Elf code more readable. This highlighting can use different colors and faces. Unfortunately, the necessary libraries are at present not standardized between XEmacs and FSF Emacs, which means that highlighting support is less general and less portable than the plain Twelf mode.

At present, highlighting has not been extensively tested in various versions of Emacs, but the font-lock mode provided in '`emacs/twelf-font.el`' seems to work at least in XEmacs version 19.16 and FSF Emacs version 19.34. The alternative highlight mode provided in '`emacs/twelf-hilit`' appears to work in FSF Emacs 19.34.

Unlike other font-lock modes, Twelf's fontification is not 'electric' in that it does not fontify as one types. One has to explicitly issue a command to fontify the current Twelf declaration or current buffer, since single-line highlighting is too error-prone and multi-line immediate highlighting is not well supported in current versions of font lock mode.

`C-c C-l`
`M-x twelf-font-fontify-decl`
: Fontifies the current Twelf declaration.

`C-c l`
`M-x twelf-font-fontify-buffer`
: Fontitifies the current buffer as Twelf code

`M-x twelf-font-unfontify`
: Removes fontification from current buffer.

12.14 Emacs Initialization

If Twelf has been properly installed, you can use the Twelf's Emacs interface with the default settings simply by adding the line

```
(load "directory/emacs/twelf-init.el")
```

to your '`.emacs`' file, where *directory* is the Twelf root directory. In order to customize the behavior, you might copy the file '`emacs/twelf-init.el`' or its contents and change it as appropriate.

12.15 Command Summary

```
--- Editing Commands ---
TAB          twelf-indent-line
DEL          backward-delete-char-untabify
M-C-q        twelf-indent-decl

--- Type Checking ---
C-c C-c      twelf-save-check-config
C-c C-s      twelf-save-check-file
C-c C-d      twelf-check-declaration
C-c c        twelf-type-const
C-c C-u      twelf-server-display

--- Error Tracking ---
C-c `        twelf-next-error
C-c =        twelf-goto-error

--- Syntax Highlighting ---
C-c C-l      twelf-font-fontify-decl
C-c l        twelf-font-fontify-buffer

--- Server State ---
C-c <        twelf-set
C-c >        twelf-get
C-c C-i      twelf-server-interrupt
M-x twelf-server
M-x twelf-server-configure
M-x twelf-server-quit
M-x twelf-server-restart
M-x twelf-server-send-command

--- Info ---
C-c C-h      twelf-info

--- Timers ---
M-x twelf-timers-reset
M-x twelf-timers-show
M-x twelf-timers-check

--- Tags (standard Emacs etags package) ---
M-x twelf-tag
M-.          find-tag (standard binding)
C-x 4 .      find-tag-other-window (standard binding)
C-c q        tags-query-replace (Twelf mode binding)
C-c s        tags-search (Twelf mode binding)
M-,          tags-loop-continue (standard binding)
             visit-tags-table, list-tags, tags-apropos
```

```
--- Communication with inferior Twelf-SML process (not Twelf Server) ---
M-x twelf-sml
C-c C-e      twelf-sml-send-query
C-c C-r      twelf-sml-send-region
C-c RET      twelf-sml-send-newline
C-c ;        twelf-sml-send-semicolon
C-c d        twelf-sml-cd
M-x twelf-sml-quit

--- Variables ---
twelf-indent
```

13 Installation

At present, Twelf has been tested in SML of New Jersey (version 110 or higher) and MLWorks, both of which implement Standard ML (revised 1997) and the Standard ML Basis Library. The instructions below apply to a Unix system. For instructions for other architectures or updates please check the file '`INSTALL`' at the Twelf home page and in the Twelf root directory after unpacking the distribution.

On a Unix system you unpack the sources with

```
gunzip twelf-1-3.tar.gz
tar -xf twelf-1-3.tar
cd twelf
make
```

This builds the Twelf server (see Chapter 11 [Twelf Server], page 65) for your current architecture and makes it accessible as '`bin/twelf-server`'. It also installs the Twelf Emacs interface (see Chapter 12 [Emacs Interface], page 69), but you must add a line

```
(load "directory/emacs/twelf-init.el")
```

to your '`.emacs`' file, where *directory* is the root directory into which you installed Twelf. Note that the Twelf installation cannot be moved after it has been compiled with `make`, since absolute pathnames are built into the executable scripts.

Note that the Twelf server presently only works with Standard ML of New Jersey, since interrupt handling is implementation specific.

If you would like to use Twelf as a structure in SML, you can then call

```
make twelf-sml
```

which creates '`bin/twelf-sml`' for the Twelf-SML mode (see Section 12.11 [Twelf-SML Mode], page 75). Calling `make clean` will remove temporary files created by the SML compiler, but not the executable file.

SML of New Jersey (free, version 110 or higher)
See `http://cm.bell-labs.com/cm/cs/what/smlnj/index.html`

MLWorks (commercial)

See `http://www.harlequin.com/products/ads/ml/ml.html`

In MLWorks, you can presently only directly load the Twelf sources, using the file '`load.sml`'.

`mlworks-basis`	start MLWorks with basis library in Twelf root directory
`use "load.sml";`	compile and load Twelf

14 Examples

We give here only a brief reference to the examples in the '`examples/`' subdirectory of the distribution. Each example comes in a separate subdirectory whose name is listed below.

'`arith`' Associativity and commutative of unary addition.

'`ccc`' Cartesian-closed categories (currently incomplete).

'`church-rosser`'
The Church-Rosser theorem for untyped lambda-calculus.

'`compile`' Various compilers starting from Mini-ML.

'`cut-elim`'
Cut elimination for intuitionistic and classical logic.

'`fol`' Simple theorems in first-order logic.

'`guide`' Examples from Users' Guide.

'`incll`' Logic programming in ordered logic.

'`kolm`' Double-negation interpretation of classical in intuitionistic logic.

'`lp`' Logic programming, uniform derivations.

'`lp-horn`' Horn fragment of logic programming.

'`mini-ml`' Mini-ML, type preservation and related theorems.

'`polylam`' Polymorphic lambda-calculus.

'`prop-calc`'
Natural deduction and Hilbert propositional calculus

'`units`' Mini-ML extended with units (currently incomplete).

In each directory or subdirectory you can find a file '`sources.cfg`' which defines the standard configuration, usually just the basic theory. The '`test.cfg`' which also defines an extended configuration with some test queries and theorems. Most examples also have a '`README`' file with a brief explanation and pointer to the literature.

15 History

While the underlying type theory has not changed, the Twelf implementation differs from older Elf implementation in a few ways. Mostly, these are simplifications and improvements. The main feature which has not yet been ported is the Elf server interface to Emacs. Also, while the type checker is more efficient now, the operational semantics does not yet incorporate some of the optimizations of the older Elf implementations and is therefore slower.

Syntax (see Chapter 3 [Syntax], page 7)

The quote '`'`' character is no longer a special character in the lexer, and '`=`' (equality) is now a reserved identifier. The syntax of `%name` declarations has changed by allowing only one preferred name to be specified. Also, `%name`, `%infix`, `%prefix` and `%postfix` declarations must be terminated by a period '`.`' which previously was optional. Further, single lines comments now must start with '`%`*whitespace*' or '`%%`' in order to avoid misspelled keywords of the form '`%`*keyword*' to be ignored.

Type theory

Elf 1.5 had two experimental features which are not available in Twelf: polymorphism and the classification of *type* as a type.

Definitions (see Section 3.3 [Definitions], page 9)

Twelf offers definitions which were not available in Elf.

Searching for definitions (see Section 5.2 [Solve Declaration], page 20)

Elf had a special top-level query form `sigma [x:A] B` which searched for a solution `M : A` and then solved the result of subsituting `M` for `x` in `B`. In Twelf this mechanism has been replaced by a declaration `%solve c : A` which searches for a solution `M : A` and then defines `c = M : A`, where the remaining free variables are implicitly universally quantified.

Query declarations (see Section 5.1 [Query Declaration], page 19)

Twelf allows queries in ordinary Elf files as '`%query`' declarations. Queries are specified with an expected number of solutions, and the number of solutions to search for, which can be used to test implementations.

Operational semantics (see Section 5.5 [Operational Semantics], page 23)

Twelf eliminates the distinction between static and dynamic signatures. Instead, dependent function types `{x:A} B` where `x` occurs in the normal form of `B` are treated statically, while non-dependent function type `A -> B` or `B <- A` or `{x:A} B` where `x` does not occur in `B` are treated dynamically.

Modes (see Chapter 7 [Modes], page 37)

Twelf offers a mode checker which was only partially supported in Elf.

Termination (see Chapter 8 [Termination], page 41)
: Twelf offers a termination checker which can verify that certain programs represent decision procedures.

Theorem prover (see Chapter 9 [Theorem Prover], page 49)
: Although very limited at present, an experimental prover for theorems and meta-theorems (that is, properties of signatures) is now available. It does not yet support lemmas or meta-hypothetical reasoning, which are currently under development.

Emacs interface (see Chapter 12 [Emacs Interface], page 69)
: The Elf mode has remained basically unchanged, but the Elf server interface has not yet been ported.

Index

F

G

H

I

K

L

M

S

T

U

V

Table of Contents