Twelf

Carsten Schürmann *
Department of Computer Science
Yale University
carsten@cs.yale.edu

June 26, 2002

The Twelf system [PS99] is a tool that is designed for experimenting with deductive systems as they occur in the theory of programming languages and logics. It supports a variety of tasks which we explain in these notes in more detail:

- *Specification* of object languages and their semantics.
- Implementation of *algorithms* manipulating object-language expressions and deductions
- Formal development of the *meta-theory* of an object language.

Several extensive experiments have been conducted with Twelf, such as the formal development of the theory of logic and functional programming languages and various logics, especially in proof carrying code architectures [Nec97, App01], and typed assembly language [MWCG99]. In these notes, we take the simply typed λ-calculus and parallel reductions as an example. One can continue the experiments sketched out below, and prove the Church-Rosser theorem for this object language — automatically.

1 Specification

Twelf employs the representation methodology and underlying type theory of the LF logical framework [HHP93]. Expressions are represented as LF objects using the technique of *higher-order abstract syntax* whereby variables of the object language are mapped to variables in the meta-language. This means that common operations, such as renaming of bound variables or capture-avoiding substitution are directly supported by the framework and do not need to be programmed anew for each object language.

For semantic specification LF uses the *judgments-as-types* representation technique. This means that a derivation is coded as an object whose type represents the judgment it establishes. Checking the correctness of a derivation is thereby reduced to type-checking its representation in the logical framework (which is efficiently decidable).

Example 1.1 (Parallel reduction) Our running example is the simply typed λ-calculus defined together with a suitable reduction semantics. The underlying notion of reduction is parallel reduction for which we can show the Church-Rosser property quite naturally.

$$\begin{array}{llll} \textit{Types:} & \tau & ::= & a \mid \tau_1 \longrightarrow \tau_2 \\ \textit{Terms:} & e & ::= & x \mid \text{lam } x : \tau.e \mid e_1 \ @ \ e_2 \end{array}$$

We write a for base types, which we will not consider further since they can be added quite easily retroactively. Function types are written as $\tau_1 \longrightarrow \tau_2$. Terms are structures that consist

*This research is funded in part by NFS under grant CCR-0133502.

of variables, λ-abstractions, and applications. In Twelf, types and terms are being represented as (LF-)types, and $\longrightarrow$, lam, and @ as constants. Variables on the other hand are represented using LF variables, which renders an additional constant unnecessary. This technique is called higher-order abstract syntax. Analogously, binders of the object languages are modeled by the binding constructs of LF.

$$\begin{array}{rcl} \text{tp} & : & \text{type} \\ \text{arrow} & : & \text{tp} \to \text{tp} \to \text{tp} \\ \\ \text{term} & : & \text{tp} \to \text{type} \\ \text{lam} & : & (\text{term}\ T_1 \to \text{term}\ T_2) \to \text{term}\ (T_1\ \text{arrow}\ T_2) \\ \text{app} & : & \text{term}\ (T_2\ \text{arrow}\ T_1) \to \text{term}\ T_2 \to \text{term}\ T_1 \end{array}$$

In this example, we use the standard trick of interpreting each uppercase variable T_1 and T_2 as universally quantified. The reason why we can leave the recovery of this kind of type information to the type inference algorithm is because their position as arguments to "term" already determines uniquely their types. Therefore, the universal closure can be automatically and quickly computed [Pfe91], relieving the user from strenuous and boring repetition of the same arguments. Subsequently, it also simplifies the use of those constants, because again, the Twelf type reconstruction algorithm will reconstruct omitted arguments. Note the difference between the function arrow $\longrightarrow$ which is part of the object language and the LF function arrow $\to$. "type" stands for the standard LF type.

The connection between the two formal languages can be easily defined by a representation function $\ulcorner\cdot\urcorner$ for which we only give one case here.

$$\ulcorner \text{lam}\ x : \tau.e \urcorner = \text{lam}\ (\lambda x : \text{term}\ \ulcorner\tau\urcorner.\ulcorner e\urcorner)$$

This particular way of encoding terms is adequate, which means that there exists a bijection between well-typed expression and their representations in LF. Incidentally it also defines a natural way of encoding and applying substitutions.

Semantically we would like to consider two terms equivalent, if they reduce to the same value via the reduction relation. We write $e_1 \implies e_2$ to express that term e_1 parallel reduces to term e_2.

$$\frac{\begin{array}{c}\overline{}\,u \\ x \implies x \\ \vdots \\ e_1 \implies e_1' \qquad e_2 \implies e_2'\end{array}}{(\lambda x : \tau.e_1)\ e_2 \implies e_1'[e_2'/x]}\ \mathsf{pbeta}^{x,u} \qquad \frac{\begin{array}{c}\overline{}\,u \\ x \implies x \\ \vdots \\ e \implies e'\end{array}}{\lambda x : \tau.e \implies \lambda x : \tau.e'}\ \mathsf{plam}^{x,u}$$

$$\frac{e_1 \implies e_1' \qquad e_2 \implies e_2'}{e_1\ e_2 \implies e_1'\ e_2'}\ \mathsf{papp}$$

$e_1 \implies e_2$ is a hypothetical judgment that is represented in LF as a type family with two index objects $\ulcorner e_1\urcorner$, and $\ulcorner e_2\urcorner$ following the judgments as types paradigm. The representation is straightforward. The reader is asked to pay special attention to the way how the hypothetical premisses of pbeta and plam are mapped into LF. Both rules discharge the assumptions x and u, which allows us to use higher level functions to represent these premisses.

$$
\begin{array}{rcl}
\Longrightarrow & : & \mathrm{term}\ T \to \mathrm{term}\ T \to \mathrm{type}. \\
\mathrm{pbeta} & : & (\Pi x : \mathrm{term}\ T.\ x \implies x \to E_1\ x \implies E_1'\ x) \\
 & & \to E_2 \implies E_2' \\
 & & \to (\mathrm{app}\ (\mathrm{lam}\ E_1)\ E_2) \implies E_1'\ E_2' \\
\mathrm{papp} & : & E_1 \implies E_1' \\
 & & \to E_2 \implies E_2' \\
 & & \to (\mathrm{app}\ E_1\ E_2) \implies (\mathrm{app}\ E_1'\ E_2') \\
\mathrm{plam} & : & (\Pi x : \mathrm{term}\ T.\ x \implies x \to E\ x \implies E'\ x) \\
 & & \to \mathrm{lam}\ E \implies \mathrm{lam}\ E'
\end{array}
$$

To improve readability, we use $\Longrightarrow$ always in infix notation. Observe how the use of higher-order functions that make the representation of the three rules above elegant, direct, and correct. The Π that is used in the signature denotes the dependent function type constructor of LF.

2 Algorithms

Generally, specification is followed by implementation of algorithms manipulating expressions or derivations. Twelf supports the implementation of such algorithms in two ways.

First, by a constraint logic programming interpretation of LF signatures, a slight variant of the one originally proposed in [Pfe91] and implemented in Elf [Pfe94]. The operational semantics is based on goal-directed, backtracking search for an object of a given type.

Example 2.1 (Logic programming) Consider the following logic programming that is written in Twelf, and uses Twelf's features, such as the dynamic introduction of parameters during runtime.

$$
\begin{array}{rcl}
\mathrm{id} & : & \Pi E : \mathrm{term}\ T.\ E \implies E \to \mathrm{type}. \\
\mathrm{id\_lam} & : & \mathrm{id}\ (\mathrm{lam}\ E)\ (\mathrm{plam}\ D) \\
 & & \leftarrow (\Pi x : \mathrm{term}\ T.\ \Pi u : x \implies x.\ \mathrm{id}\ x\ u \to \mathrm{id}\ (E\ x)\ (D\ x\ u) \\
\mathrm{id\_app} & : & \mathrm{id}\ (\mathrm{app}\ E1\ E_2)\ (\mathrm{papp}\ D_1\ D_2) \\
 & & \leftarrow \mathrm{id}\ E_1\ D_1 \\
 & & \leftarrow \mathrm{id}\ E_2\ D_2
\end{array}
$$

There are two natural ways of reading this signature. One is purely type theoretic: The type family `id` is defined together with two constants. For this interpretation, simply read the types in backward fashion following the directions of the arrow. The other way is as a logic program. In this fashion each $\leftarrow$ introduces a new subgoal and each Π introduces dynamically new parameters. In that, Twelf logic programs are very similar to λProlog logic programs. The forward pointing $\to$ extends the set of clauses dynamically during runtime, that may be applied later on.

The second way of specifying algorithms in Twelf is in form of functional programs written in the programming language Delphin [SFL02]. Programming in Delphin is like programming in ML except that the objects manipulated are instances of LF type theory, and not instances of standard datatypes.

Example 2.2 (Identity function in Delphin) The same example from above depicted in Delphin would have the following form

```
%block L : SOME {T : tp} BLOCK {x:term T} {u : x ⟹ x}.
id :: worlds (L)
        all* {T : tp} all {E : term T}
        exists {D : E ⟹ E} true.
fun id (x(x)) = u(x)
  | id (lam E) =
    let
      val <D, <>> = new x:L. id (E (x(x)))
    in
      <plam D, <>>
    end
  | id (app E1 E2) =
    let
      val <D1, <>> = id E1
      val <D2, <>> = id E2
    in
      <papp D1 D2, <>>
    end
```

Note, that instances of type $T : \text{tp}$, the first argument to `id`, can be inferred from other arguments and may therefore be omitted. The user may communicate this to Twelf by appending a `*` to the `all` quantifier.

Delphin programmers can program with LF objects as first-class objects. This means that functions may be defined by pattern-matching and recursion, just as `id` above. What is different from standard functional programming languages is that Delphin programs can dynamically introduce new parameters for types which are organized in *blocks*. Blocks are defined by the `%block` directive, a description that ties together which parameters may be introduced dynamically and what is to be done once they are encountered during execution. The `SOME` part of such a description quantifies all existential variables that may occur free in the block. Delphin programs may be parametrized by several different kinds of blocks, a collection which we call *world*. Each individual Delphin program must be declared in a specific world. Naturally, if one program is allowed to call another depends on if their worlds are compatible. In the example above, "`all`" stands for Delphin's dependent function type, "`ex`" for its dependent product type, and "`true`" is simply "`unit`" in ML.

The function `id` is defined by cases over instances of a variable $E : \text{term } T$ for some T. Three cases are possible. First E may be one of the parameters "$x(\underline{x})$". Here, $\underline{x}$ is a variable that ranges over instances of the block [Sch01]. Second, E may be a λ-term "lam E", and third, it may be an application "app E_1 E_2". The "`new`" command used in the second case allows programmers to introduce new parameters during runtime, all of which are appropriately abstracted eventually. This means that all additional parameters contained in $(\underline{x})$ are reflected down onto the LF level, which turns D into a possibly higher-level function type. The angle brackets `<>` read as unit program of type `true`. Along the same lines, $\texttt{<}D_1\texttt{, <>>}$ is a value of a dependent product type.

The implementation of Delphin is still in an experimental stage. Nevertheless, we hope that it is ready at the time of the summer school.

3 Meta-Theory

Twelf provides two related means to express the meta-theory of deductive systems: higher-level judgments and the meta-logic $\mathcal{M}_2^+$.

A higher-level judgment describes a relation between derivations inherent in a (constructive) meta-theoretic proof. Using the operational semantics for LF signatures sketched above, we can

then execute a meta-theoretic proof. While this method is very general and has been used in many of the experiments, type-checking a higher-level judgment does not by itself guarantee that it correctly implements a proof. On the contrary, the operational semantics might not terminate, or might get stuck, in either case, it will return without a value.

Twelf provides therefore a set tools that helps users to analyze the hypothetical behavior of the operational semantics. The termination checker, for example, guarantees that executing logic programs will always terminate. The mode checker ensures the correct input/output behavior of logic programs, which is non-trivial in a logic programming setting because of the global properties of logic variables. Another tool is the coverage checker which checks that the logic program contains at least one applicable case that matches every possible ground input. And finally, the totality checker certifies that the logic programming interpreter that runs this program never gets stuck. Thus if the operational semantics has not computed a value yet, it will be possible for it to make progress.

Example 3.1 (Tools) We show how to verify those four properties using the logic program for `id` from Example 2.1.

Termination Checker	`%terminates` E `(id` E `_)`
Mode Checker	`%mode (id +`E `-`D`)`
Coverage Checker	`%covers (id` E D`)`
Totality Checker	`%total` E `(id` E D`)`

The first argument in the termination declaration E defines the termination order as the well-founded subterm ordering on terms. The mode declaration reads as follows: for all (`+`) ground terms E, there exists (`-`) a ground parallel reduction D. Well-modedness is a precondition for coverage which guarantees that calling `id` will make progress. It does however not verify that the logic program may not get stuck upon return a property which is called "output coverage". This is left to the `%total` declaration. Therefore, if `%total` suceeds, the logical program implements a total function. The final touches on the implementation of `%total` are still underway, we hope however to have a a version ready by the time of the summer school.

Alternatively, one can use an experimental but automatic meta-theorem proving component based on the meta-logic $\mathcal{M}_2^+$ for LF [Sch00]. It expects as input a statement about LF objects over a fixed signature possibly open in regularly formed contexts whose structure is defined a priori. In the current version proofs are by induction which require the user to specify over which variable induction is to be performed. Induction orders are lexicographic and simultaneous extensions of the subterm ordering. Once a proof is found it can be produced either as a Delphin function, or as a representation as a higher-level judgment, which can then be executed.

Example 3.2 (Theorem Proving) The following two declarations

```
%theorem identity: worlds (L)
  all* {T : tp} all {E : term T} exists {D : E ⟹ E} true.
%prover 5 E (identity E).
```

ask Twelf to prove the theorem `identity`. Internally, the resulting proof is a total Delphin function. The 5 signals Twelf to search only for objects that are less than 5 levels deep counting constructors. The E simply states that induction goes on E.

4 Environment

While Twelf is implemented in ML it is executed as a stand-alone program rather than within the ML top-level loop. The most effective way to interact with Twelf is as an inferior process to Emacs.

There are two Emacs interfaces, one is house made, the other is Proof General. The house made Emacs interface provides an editing mode for Twelf source files and commands for incremental type checking, logic program execution, termination, mode, coverage, and totality checking, and theorem proving. Moreover it provides utilities for jumping to error locations and tagging and maintaining configurations of source files.

References

[App01] Andrew W. Appel. Foundational proof-carrying code. In *16th Annual IEEE Symposium on Logic in Computer Science (LICS '01)*, pages 247–258, Boston, USA, June 2001.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[MWCG99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.

[Nec97] George C. Necula. Proof-carrying code. In Neil D. Jones, editor, *Conference Record of the 24th Symposium on Principles of Programming Languages (POPL'97)*, pages 106–119, Paris, France, January 1997. ACM Press.

[Pfe91] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[Pfe94] Frank Pfenning. Elf: A meta-language for deductive systems. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, pages 811–815, Nancy, France, June 1994. Springer-Verlag LNAI 814. System abstract.

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

[Sch00] Carsten Schürmann. *Automating the Meta-Theory of Deductive Systems*. PhD thesis, Carnegie Mellon University, 2000. CMU-CS-00-146.

[Sch01] Carsten Schürmann. Recursion for higher-order encodings. In Laurent Fribourg, editor, *Proceedings of the Conference on Computer Science Logic (CSL 2001)*, pages 585–599, Paris, France, August 2001. Springer Verlag LNCS 2142.

[SFL02] Carsten Schürmann, Richard Fontana, and Yu Liao. Delphin: Functional programming with deductive systems. Submitted, 2002.