A Practical Module System for LF

Florian Rabe, Carsten Schürmann

Jacobs University Bremen, IT University Copenhagen

History

- Harper, Honsell, Plotkin, 1993: LF
- Harper, Pfenning, 1998: A Module System [... for] LF
- Pfenning, Schürmann, 1999: Twelf (implementation)
- Watkins, 2001: A simple module language for LF (partially integrated into Twelf)
- Licata, Simmons, Lee, 2006: A simple module system for Twelf (stand-alone implementation)
- Rabe, 2008: language-independent module system (stand-alone implementation)
- Rabe, Schürmann, 2009: instantiation of above with LF (integrated into Twelf)

Design goals

- Name space management
- Code reuse
- No effects on the underlying theory
- Modular proof design

```
%sig IProp = {
  o    : type.
  imp  : o → o → o.
  not  : o → o.
  true : o → type.
  impI, impE, notI, notE : ...
}.
%sig CProp = {
  prop : type.
  ded  : o → type.
  %struct I : IProp = {o := prop. true := ded.}.
  dne  : ded ((I.not I.not A) I.imp A).
}.
```

Primitive Concepts and Examples

Running Example

1. Monoid is a signature declaring a base type and operations on it.
2. List is a signature that takes an arbitrary monoid M and declares the type of list over M.
3. Lists over a monoid can be folded.
4. The natural numbers are a monoid under addition.
5. Using the above, we can compute $\mathit{fold}(1 :: 1 :: \mathit{nil}) = 2$.

Signatures and Structures

Signatures are collections of declarations:

```
%sig Monoid = {
  a    : type.
  unit : a.
  comp : a → a → a → type.
}.
```

Structures instantiate signatures:

```
%sig List = {
  %struct elem : Monoid
  list       : type.
  nil        : list.
  cons       : elem.a → list → list.
  fold       : list → elem.a → type.
  foldnil    : fold nil elem.unit.
  foldcons   : fold L B → elem.comp A B C
             → fold (cons A L) C.
}.
```

Signatures and Views

Signatures unify interfaces ...

```
%sig Monoid =
  {a : type. unit : a. comp : a → a → a → type.}.
```

... and implementations:

```
%sig Nat = {
  nat      : type.
  zero     : nat.
  succ     : nat → nat.
  add      : nat → nat → nat → type.
  addzero  : add N zero N.
  addsucc  : add N P Q → add N (succ P) (succ Q).
}.
```

Views connect signatures:

```
%view NatMonoid : Monoid → Nat = {
  a     := nat.
  unit  := zero.
  comp  := add.
}.
```

Instantiations

Seen so far:

```
%sig Monoid = {...}.
%sig List   = {%struct elem : Monoid. ...}.
%sig Nat    = {...}.
%view NatMonoid : Monoid → Nat = {...}.
```

Instantiations provide values for parameters:

```
%struct nat : Nat.
%struct l   : List = {
  %struct elem :=          nat.
}.
```

Then $\mathit{fold}(1 :: 1 :: \mathit{nil}) = 2$ is computed by:

```
%solve _ : l.fold (l.cons (nat.succ nat.zero)
                  (l.cons (nat.succ nat.zero) l.nil)
                  ) N.
N = nat.succ (nat.succ nat.zero).
```

Instantiations

Seen so far:

```
%sig Monoid = {...}.
%sig List   = {%struct elem : Monoid.  ...}.
%sig Nat    = {...}.
%view NatMonoid : Monoid → Nat = {...}.
```

Instantiations provide values for parameters:

```
%struct nat : Nat.
%struct l   : List = {
  %struct elem := NatMonoid nat.
}.
```

Then $fold(1 :: 1 :: nil) = 2$ is computed by:

```
%solve _ : l.fold (l.cons (nat.succ nat.zero)
                  (l.cons (nat.succ nat.zero) l.nil)
                  ) N.
N = nat.succ (nat.succ nat.zero).
```

Type System

General Idea

1. Determine elaborated declarations available in a given signature (10 rules)
2. Reuse LF typing for objects, define typing for morphisms (LF plus 7 rules)
3. Define modular signatures using the above (9 rules)

$$\frac{T \ = \ \{\ldots,\ c : A = B,\ \ldots\} \text{ in } \mathcal{G}}{\mathcal{G} \ggg_T c : A = B} \qquad \frac{T \ = \ \{\ldots,\ c : A,\ \ldots\} \text{ in } \mathcal{G}}{\mathcal{G} \ggg_T c : A}$$

$$\frac{\mathcal{G} \ggg T\text{''}s : S \to T = \_ \quad \mathcal{G} \ggg_S \vec{c} : A = B \quad \mathcal{G} \ggg_{T\text{''}s} \vec{c} := B'}{\mathcal{G} \ggg_T s.\vec{c} : T\text{''}s(A) = B'}$$

$$\frac{\mathcal{G} \ggg T\text{''}s : S \to T = \_ \quad \mathcal{G} \ggg_S \vec{c} : A = B \quad \mathcal{G} \ggg_{T\text{''}s} \vec{c} := \bot}{\mathcal{G} \ggg_T s.\vec{c} : T\text{''}s(A) = T\text{''}s(B)}$$

Figure: Elaboration

$$\frac{\mathcal{G} \ggg_T \vec{c} : A = \_}{\mathcal{G} \vdash_T \vec{c} : A}\,\mathcal{T}_{:} \qquad \frac{\mathcal{G} \ggg_T \vec{c} : \_ = B,\; B \neq \bot}{\mathcal{G} \vdash_T \vec{c} \equiv B}\,\mathcal{T}_{\equiv}$$

$$\frac{\mathcal{G} \ggg m : S \to T = \_}{\mathcal{G} \vdash m : S \to T}\,\mathcal{M}_m$$

$$\frac{\mathcal{G} \vdash \mu : R \to S \quad \mathcal{G} \vdash \mu' : S \to T}{\mathcal{G} \vdash \mu\,\mu' : R \to T}\,\mathcal{M}_{comp}$$

Figure: Typing

Results and Discussion

Conservativity

```
%sig Monoid = {
  a    : type.
  unit : a.
  comp : a → a → a → type.
}.
```

Modular signatures are elaborated to non-modular signatures:

Modular

```
%sig List = {
  %struct elem : Monoid.




  list : type.
  ...
}.
```

Non-modular

```
List"elem.a    : type.
List"elem.unit : List"elem.a.
List"elem.comp : List"elem.a
               → List"elem.a
               → List"elem.a
               → type.
List"list      : type.
...
```

Theorem: Elaborated signature is well-formed iff modular one is.

Signature Morphism Semantics

- Morphism from S to T: type-preserving structural/homomorphic/recursive map of S-objects to T-objects
- View from S to T: concrete syntax for signature morphism
- Structure of type S within signature T: induces signature morphism from S to T
- Theorem: instantiation `%struct` $s := M$ in m implies $e \circ s \equiv m$.

```
%sig Monoid = {...}.
%sig List   = {
 %struct elem: Monoid ...}.
%sig Nat    = {...}.
%view NatMonoid :
  Monoid -> Nat = {...}.
%struct nat : Nat.
%struct l   : List = {
  %struct elem:=NatMonoid nat.}.
```

$$\begin{tikzcd}
 & \texttt{List} \arrow[dr, "\texttt{l}"] & \\
\texttt{Monoid} \arrow[ur, "\texttt{elem}"] \arrow[dr, "\texttt{NatMonoid}"'] & & \texttt{Toplevel} \\
 & \texttt{Nat} \arrow[ur, "\texttt{nat}"'] &
\end{tikzcd}$$

Implementation

- One full-time researcher month, daily meetings with Carsten
- Design and major implementation decisions fixed a priori
- Partial reuse of Watkins's parser and lexer
- One week for changing Twelf's core data structures
- Current state:
  - LF aspects fully implemented, tested, documented, case studies done, ready to merge into trunk
  - All features of non-modular Twelf preserved
  - Modular Twelf aware of fixity, name, mode declarations
  - Modular Twelf not aware of meta-theory yet

Case Studies

- Logic: Modular design of classical and intuitionistic logic and Kolmogoroff translation for each connective [Rabe, Schürmann]
- Logic: Modular design of first-order logic – syntax, proof theory, set-theoretic semantics, soundness for each connective/quantifier [Horozal, Rabe] (1300 LOC)
- Type theory: Modular design of type theories following the lambda cube [Horozal, Rabe]
- Programming: Modular design of Mini-ML and modularized coverage proofs [Schürmann]
- Algebra: monoids, ..., fields, orders, ..., lattices [Dumbrava, Horozal, Sojakova] (600 LOC)

Discussion

- Why is feature X missing? deliberately simple design
- Why views?
  generalization of structural subtyping, fitting morphisms
- What about functors?
  generalized views intended to subsume functors
- What about the Twelf meta-theory?
  still a theoretical challenge

Conclusion

- Finally a working module system as part of Twelf
- Fully conservative: modular signatures are elaborated to non-modular ones, non-modular signatures type-check as before
- Modular structure preserved during type-checking
- Future work: Twelf meta-theory — feedback needed
- Homepage: `http://www.twelf.org/mod/`
- SVN: `https://cvs.concert.cs.cmu.edu/twelf/branches/twelf-mod`

  to be merged into trunk soon

Structures and Views

	Structures	Views
action	induced	explicitly given
morphism property	by definition	by type-checking
relating signatures	inheritance	translation/realization
signature subtyping	nominal	structural

```
%sig Monoid={a: type ...}.
%sig Group={
 %struct mon: Monoid.
 ...
}.
```

```
%sig Nat={nat: type ...}.
%view NatMonoid:
 Monoid->Nat={a:= nat ...}
    .
```