A Practical Module System for LF

Florian Rabe, Carsten Schürmann

Jacobs University Bremen, IT University Copenhagen
History

- 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

\%
\textbf{sig} \quad \text{IProp} = \{ \\
\quad \ o \ : \ \text{type} . \\
\quad \ \text{imp} \ : \ o \ \to \ o \ \to \ o . \\
\quad \ \text{not} \ : \ o \ \to \ o . \\
\quad \ \text{true} \ : \ o \ \to \ \text{type} . \\
\quad \ \text{impl} , \ \text{impE} , \ \text{notI} , \ \text{notE} : \ldots \\
\}\text{.}

\%
\textbf{sig} \quad \text{CProp} = \{ \\
\quad \ \text{prop} \ : \ \text{type} . \\
\quad \ \text{ded} \ : \ o \ \to \ \text{type} . \\
\%
\textbf{struct} \quad \text{l} : \ \text{IProp} = \{ \text{o} := \ \text{prop} . \ \text{true} := \ \text{ded} . \} . \\
\quad \ \text{dne} : \ \text{ded} \ ((\text{l}. \ \text{not} \ \text{l}. \ \text{not} A) \ \text{l}. \ \text{imp} A) . \\
\}\text{.}
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 $\text{fold}(1 :: 1 :: \text{nil}) = 2$. 
Signatures and Structures

Signatures are collections of declarations:

\[
\begin{align*}
\%\text{sig } \text{Monoid} &= \{ \\
    &\quad \text{a : type.} \\
    &\quad \text{unit : a.} \\
    &\quad \text{comp : a }\to\text{ a }\to\text{ a }\to\text{ type.} \\
\};
\end{align*}
\]

Structures instantiate signatures:

\[
\begin{align*}
\%\text{sig } \text{List} &= \{ \\
    &\quad \%\text{struct } \text{elem : Monoid} \\
    &\quad \text{list : type.} \\
    &\quad \text{nil : list.} \\
    &\quad \text{cons : elem }\cdot\text{a }\to\text{ list }\to\text{ list.} \\
    &\quad \text{fold : list }\to\text{ elem }\cdot\text{a }\to\text{ type.} \\
    &\quad \text{fold\,nil : fold\,nil\,elem\,unit.} \\
    &\quad \text{fold\,cons : fold\,L\,B }\to\text{ elem\,comp\,A\,B\,C} \\
    &\quad \quad \to\text{ fold\,(cons\,A\,L)\,C.}
\};
\end{align*}
\]
Signatures and Views

Signatures unify interfaces ...

\%
\texttt{sig} \ Monoid =
\{\texttt{a} : \texttt{type}. \texttt{unit} : \texttt{a}. \texttt{comp} : \texttt{a} \rightarrow \texttt{a} \rightarrow \texttt{a} \rightarrow \texttt{type.}\\}

... and implementations:

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

Views connect signatures:

\%
\texttt{view} \ NatMonoid : \texttt{Monoid} \rightarrow \texttt{Nat} = \{
  \texttt{a} := \texttt{nat}. \\
  \texttt{unit} := \texttt{zero}. \\
  \texttt{comp} := \texttt{add}.
\}.
Instantiations

Seen so far:

\%sig Monoid = \{ \ldots \}.
\%sig List = \{ \%struct elem : Monoid. \ldots \}.
\%sig Nat = \{ \ldots \}.
\%view NatMonoid : Monoid \rightarrow Nat = \{ \ldots \}.

Instantiations provide values for parameters:

\%struct nat : Nat.
\%struct l : List = {
  \%struct elem := 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).
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 \( \text{fold}(1 :: 1 :: \text{nil}) = 2 \) is computed by:

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

\( N = \text{nat.succ (nat.succ nat.zero)}. \)
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)
\[
T = \{\ldots, \ c : A = B, \ldots\} \text{ in } \mathcal{G} \quad T = \{\ldots, \ c : A, \ldots\} \text{ in } \mathcal{G}
\]
\[
\mathcal{G} \gg T \ c : A = B
\]
\[
\mathcal{G} \gg T \ c : A
\]
\[
\mathcal{G} \gg T'' s : S \rightarrow T = \_ \quad \mathcal{G} \gg_{S} \vec{c} : A = B \quad \mathcal{G} \gg_{T'' s} \vec{c} : = B'
\]
\[
\mathcal{G} \gg_{T} s.\vec{c} : T'' s(A) = B'
\]
\[
\mathcal{G} \gg T'' s : S \rightarrow T = \_ \quad \mathcal{G} \gg_{S} \vec{c} : A = B \quad \mathcal{G} \gg_{T'' s} \vec{c} : = \bot
\]
\[
\mathcal{G} \gg_{T} s.\vec{c} : T'' s(A) = T'' s(B)
\]

**Figure:** Elaboration
\[
\begin{align*}
G \ggg_T \tilde{c} : A &\quad \vdash_T \quad \frac{G \ggg_T \tilde{c} : A}{G \vdash_T \tilde{c} : A} \\
G \ggg_T \tilde{c} : \_ &\quad \vdash_{=} \quad \frac{G \ggg_T \tilde{c} : \_ = B, \ B \neq \bot}{G \vdash_T \tilde{c} \equiv B} \\
G \ggg_T m : S \rightarrow T &\quad \vdash_{=} \quad \frac{G \ggg_T m : S \rightarrow T}{G \vdash m : S \rightarrow T} \quad \mathcal{M}_m \\
G \vdash \mu : R \rightarrow S &\quad G \vdash \mu' : S \rightarrow T \quad \vdash_{=} \quad \frac{G \vdash \mu \ \mu' : R \rightarrow T}{G \vdash \mu \ \mu' : R \rightarrow T} \quad \mathcal{M}_{\text{comp}}
\end{align*}
\]

**Figure:** Typing
Results and Discussion
Conservativity

\%sig Monoid = {
  a : type.
  unit : a.
  comp : a \rightarrow a \rightarrow a \rightarrow 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 \rightarrow List"elem"a \rightarrow List"elem"a \rightarrow 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 $\%\text{struct } s := M \text{ in } m$ 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.}.

\[ e \circ s \equiv m. \]
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

<table>
<thead>
<tr>
<th></th>
<th>Structures</th>
<th>Views</th>
</tr>
</thead>
<tbody>
<tr>
<td>action</td>
<td>induced by definition</td>
<td>explicitly given by type-checking</td>
</tr>
<tr>
<td>morphism property</td>
<td>by definition inheritance</td>
<td>translation/realization</td>
</tr>
<tr>
<td>relating signatures</td>
<td>nominal</td>
<td>structural</td>
</tr>
<tr>
<td>signature subtyping</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
%\text{sig } \text{Monoid}=\{a: \text{type} \ldots \}. \\
%\text{sig } \text{Nat}=\{\text{nat: type} \ldots \}. \\
%\text{struct } \text{mon}: \text{Monoid}. \\
%\text{view } \text{NatMonoid}: \text{Monoid}\rightarrow\text{Nat}=\{a:=\text{nat} \ldots \}
\]