A Translation-Based Animation of Dependently-Typed Specifications

From LF to *hohh*(and back again)

Mary Southern and Gopalan Nadathur

Department of Computer Science and Engineering
University of Minnesota

This work was funded by NSF grant CCF-0917140.

Some Motivation

We are interested in formalizing systems that are described in a rule-based and syntax directed fashion

Two approaches with complementary benefits exist for formalizing such systems:

- An approach based on using dependently-typed λ-calculi

  Primary Virtue: Dependent types are a convenient and widely used means for encoding specifications
- An approach that uses logical predicates over λ-calculus terms

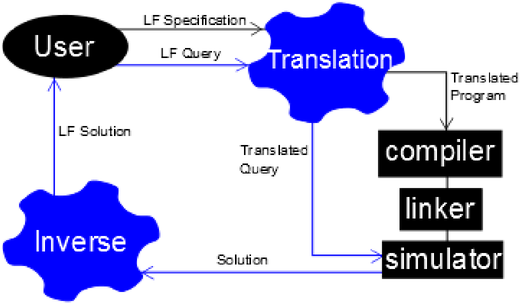
  Primary Virtue: Such a logic has an efficient implementation and specifications in it can also be expressively reasoned about

Our Goal: To harness the benefits of *both* approaches

Specifically, we want to

- let the first approach be used for developing specifications
- use a translation to the second form to realize animation

Map of Talk


User
LF Specification
LF Query
Translation
Translated
Program
LF Solution
Translated
Query
compiler
linker
Inverse
Solution
simulator

Edinburgh Logical Framework (LF)

Syntax of Expressions

Kind	K	$:= \mathit{Type} \mid \Pi x{:}A.K$
Type	A	$:= a\ M \ldots M \mid \Pi x{:}A.A$
Object	M	$:= c \mid x \mid X \mid \lambda x{:}A.M \mid M\ M$

We are interested in deriving judgments of the form:

$$\Gamma \vdash_{\Sigma} M : A$$

This is done with respect to:

- Signature $\Sigma := \cdot \mid \Sigma, c : A \mid \Sigma, a : K$
- Context $\Gamma := \cdot \mid \Gamma, x : A$
- Meta-Variable Context Δ

Example Specification

```
nat N := 0 | S N
list L := [] | (N ::  L)
```

$$\frac{}{L_1 \ @ \ L_2 \ = \ L_3} \qquad \frac{L_1 \ @ \ L_2 \ = \ L_3}{(X \ :: \ L_1) \ @ \ L_2 \ = \ (X \ :: \ L_3)}$$

$$\begin{array}{lll} \mathit{nat} & : & \mathit{type}. \\ z & : & \mathit{nat}. \\ s & : & \mathit{nat} \to \mathit{nat}. \end{array} \qquad \begin{array}{lll} \mathit{list} & : & \mathit{type}. \\ \mathit{nil} & : & \mathit{list}. \\ \mathit{cons} & : & \mathit{nat} \to \mathit{list} \to \mathit{list}. \end{array}$$

$$\begin{array}{lll} \mathit{app} & : & \mathit{list} \to \mathit{list} \to \mathit{list} \to \mathit{type}. \\ \mathit{app\_N} & : & \Pi L{:}\mathit{list}.\mathit{app} \ \ \mathit{nil} \ \ L \ \ L. \\ \mathit{app\_C} & : & \Pi X{:}\mathit{nat}.\Pi L_1{:}\mathit{list}.\Pi L_2{:}\mathit{list}.\Pi L_3{:}\mathit{list}. \\ & & \quad \Pi A{:}\mathit{app} \ \ L_1 \ \ L_2 \ \ L_3.\mathit{app} \ \ (\mathit{cons} \ \ X \ \ L_1) \ \ L_2 \ \ (\mathit{cons} \ \ X \ \ L_3) \end{array}$$

A Predicate Logic

- We work with a fragment of the logic of Higher-Order Hereditary Harrop Formulas (*hohh*)
- This logic underlies the logic programming language λ Prolog

Atomic formulas, A, are constructed using predicate symbols that take simply typed λ-terms as arguments.

Formulas

$$D \;:=\; A \mid G \supset D \mid \forall x.D \qquad G \;:=\; \top \mid A \mid D \supset G \mid \forall x.G$$

- A collection of D-formulas, or Program $\mathcal{P}$, encodes a specification and a G formula corresponds to a query

Logic Programming - Predicate Logic

We want to derive sequents of the form: $\Xi; \mathcal{P} \longrightarrow G$
where

- Ξ is the signature containing the term constants
- $\mathcal{P}$ is a program (set of D-formulas)
- G is the goal formula we wish to solve

Two main differences from Logic Programming in Prolog:

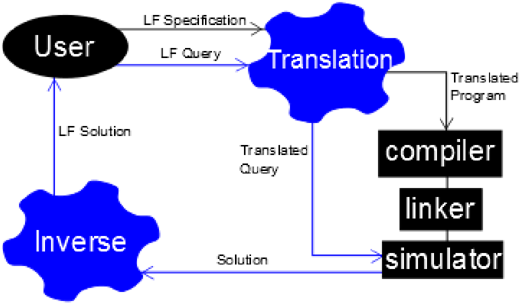
- Program can be extended dynamically

$$\frac{\Xi; \Gamma, D \longrightarrow G}{\Xi; \Gamma \longrightarrow D \supset G}$$

- Signature can be extended dynamically

$$\frac{\Xi, c; \Gamma \longrightarrow G[c/x]}{\Xi; \Gamma \longrightarrow \forall x. G}$$

Map of Talk



User
LF Specification
LF Query
Translation
Translated
Program
LF Solution
Translated
Query
compiler
linker
Inverse
Solution
simulator

Overview of Translation

The translation is based on a two step process

1. First we map *both* LF types and objects into simply typed λ-terms.
   - we use *hohh* terms of type *lf-type* for LF types
   - we use *hohh* terms of type *lf-obj* for LF objects

   Notice that the LF typing information is lost in this translation and only the functional structure of expressions is retained
2. We then encode LF typing relations in predicates over the *hohh* terms denoting LF objects and LF types
   In particular,
   - the predicate *hastype* : $\mathit{lf\text{-}obj} \rightarrow \mathit{lf\text{-}type} \rightarrow o$ is used for this.

A Translation 1/2

The encoding of LF terms, $\langle\cdot\rangle$ is given by the rules below.

$$\langle c\rangle := c \qquad \langle x\rangle := x \qquad \langle X\rangle := X$$

$$\langle M\ N\rangle := \langle M\rangle\ \langle N\rangle \qquad \langle \lambda x{:}A.M\rangle := \lambda x.\langle M\rangle$$

The mapping, $\phi(\cdot)$ flattens the types of LF terms:

$$\phi(\mathit{Type}) := \mathit{lf\text{-}type} \qquad \phi(\Pi x{:}A.B) := \phi(A) \to \phi(B)$$

$$\phi(A) := \mathit{lf\text{-}obj} \quad \text{when } A \text{ is a base type}$$

Example Encoding

$\mathit{nat} : \mathit{lf\text{-}type}.$ $\qquad$ $\mathit{list} : \mathit{lf\text{-}type}.$

$z : \mathit{lf\text{-}obj}.$ $\qquad$ $\mathit{nil} : \mathit{lf\text{-}obj}.$

$s : \mathit{lf\text{-}obj} \to \mathit{lf\text{-}obj}.$ $\qquad$ $\mathit{cons} : \mathit{lf\text{-}obj} \to \mathit{lf\text{-}obj} \to \mathit{lf\text{-}obj}.$

$\mathit{app} : \mathit{lf\text{-}obj} \to \mathit{lf\text{-}obj} \to \mathit{lf\text{-}obj} \to \mathit{lf\text{-}type}.$

$\mathit{app\_N} : \mathit{lf\text{-}obj} \to \mathit{lf\text{-}obj}.$

$\mathit{app\_C} : \mathit{lf\text{-}obj} \to \mathit{lf\text{-}obj} \to \mathit{lf\text{-}obj} \to \mathit{lf\text{-}obj} \to \mathit{lf\text{-}obj} \to \mathit{lf\text{-}obj}.$

A Translation 2/2

Then, LF types are translated as follows:

$$\{\!\{A\}\!\} \quad := \quad \lambda M.\ \mathit{hastype}\ M\ \langle A \rangle \qquad \text{if } A \text{ is a base type}$$

$$\{\!\{\Pi x{:}A.B\}\!\} \quad := \quad \lambda M.\ \forall x.\ (\{\!\{A\}\!\}\ x) \supset (\{\!\{B\}\!\}\ (M\ x))$$

For example, consider the translation of $\Pi L{:}\mathit{list}.\mathit{app}\ \mathit{nil}\ L\ L$:

$\{\!\{\Pi L{:}\mathit{list}.\mathit{app}\ \mathit{nil}\ L\ L\}\!\}$
$\lambda M.\ \forall L.\ (\{\!\{\mathit{list}\}\!\}\ L) \supset (\{\!\{\mathit{app}\ \mathit{nil}\ L\ L\}\!\}\ (M\ L))$
$\lambda M.\ \forall L.\ (\mathit{hastype}\ L\ \mathit{list}) \supset (\mathit{hastype}\ (M\ L)\ (\mathit{app}\ \mathit{nil}\ L\ L))$

Thus, the LF signature item $\mathit{app\_N} : \Pi L{:}\mathit{list}.\mathit{app}\ \mathit{nil}\ L\ L$ yields the λ-Prolog formula

$$\forall L.\ (\mathit{hastype}\ L\ \mathit{list} \supset \mathit{hastype}\ (\mathit{app\_N}\ L)\ (\mathit{app}\ \mathit{nil}\ L\ L))$$

Improving the Translation

Consider the constant app_C.

$$app\_C : \Pi X{:}nat.\Pi L_1{:}list.\Pi L_2{:}list.\Pi L_3{:}list.\Pi A{:}app\ L_1\ L_2\ L_3.$$
$$app\ (cons\ X\ L_1)\ L_2\ (cons\ X\ L_3)$$

Whenever we are matching an instance of this type, we must ensure that the terms being substituted for the Π-bound variables are of the correct type.

- Certain terms will appear in such a way that we know this to be the case.

Consider a well-formed type: $app\ (cons\ x\ l1)\ l2\ (cons\ x\ l3)$.

- Clearly then, whatever the term $l1$ (resp. $l2$, $l3$), it must be of type *list*
- Similarly x must be of type *nat*
- But is there a term of type $app\ l1\ l2\ l3$?

Characterizing Redundancies

This type checking becomes the *hastype* formula of the Π-bound variable.

By categorizing which of these checks is unnecessary, we are able to reduce the number of goals which must be satisfied during proof search.

- The essential idea is that we do not need to perform such a check when there is an occurrence whose structure is not lost or altered by other substitutions.

We define a criterion, called Strictness, which captures this idea.

Strictness

1. There is an occurrence, in the head of the type, which does not disappear after performing substitutions for the other Π-quantified variables.
2. This occurrence may only be applied to distinct λ-bound variables.

Strictness

There are two main judgments associated with strictness:

$$\Gamma; x \sqsubset_t A \qquad \text{and} \qquad \Delta; \delta; x \sqsubset_o M$$

- Γ collects Π-bound variables

- Δ contains the Π-bound variables
- δ collects λ-bound variables

Translation now proceeds in two modes:

- In the positive context we remove the *hastype* clause for strictly occurring variables.
- In the negative context we proceed as before.

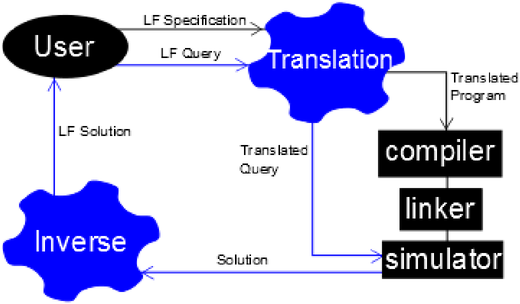
Example Specification - Translated

$$
\begin{array}{lll}
nat : \textit{lf-type}. & & list : \textit{lf-type}. \\
z : \textit{lf-obj}. & & nil : \textit{lf-obj}. \\
s : \textit{lf-obj} \to \textit{lf-obj}. & & cons : \textit{lf-obj} \to \textit{lf-obj} \to \textit{lf-obj}.
\end{array}
$$

$$
\begin{array}{l}
app : \textit{lf-obj} \to \textit{lf-obj} \to \textit{lf-obj} \to \textit{lf-type}. \\
app\_N : \textit{lf-obj} \to \textit{lf-obj}. \\
app\_C : \textit{lf-obj} \to \textit{lf-obj} \to \textit{lf-obj} \to \textit{lf-obj} \to \textit{lf-obj} \to \textit{lf-obj}.
\end{array}
$$

$$
\begin{array}{l}
\forall L.\textit{hastype}\ (app\_N\ L)\ (app\ nil\ L\ L). \\
\forall X.\forall L_1.\forall L_2.\forall L3.\forall A.\textit{hastype}\ A\ (app\ L_1\ L_2\ L_3) \supset \\
\quad \textit{hastype}\ (app\_C\ X\ L_1\ L_2\ L_3\ A)\ (app\ (cons\ X\ L_1)\ L_2\ (cons\ X\ L_3)).
\end{array}
$$

Map of Talk



Motivation

Specifications

A Translation

An Inverse

Looking Forward

Dealing with Queries

After writing an LF specification, one may want to present and solve queries of the form $M : A$.

- We allow logic variables to appear in the type A.

LF Query $\quad Proof : \Pi x{:}nat.app\ nil\ (cons\ z\ (cons\ x\ nil))\ (L\ x)$

Translated Query

$$\forall x.hastype\ Proof\ (app\ nil\ (cons\ z\ (cons\ x\ nil))\ (L\ x))$$

Solution $\quad L = \lambda y.cons\ z\ (cons\ y\ nil)$

$\quad Proof = \lambda y.app\_N\ (cons\ z\ (cons\ y\ nil))$

We would like to now return our solution to LF.
There are two concerns we should keep in mind:

- Under our chosen signature, there may be well-formed STLC terms which have no corresponding LF term.
  Eg. *arrow empty* (*app unit unit*)
- Alternatively, there may be terms with multiple corresponding LF terms.
  Eg. $(\lambda x.x)$

An Inverse Encoding

We are not interested in inverting arbitrary terms

- All terms will correspond to a well-formed LF term.
- LF typing information ensures a unique inverse.

We define the inverse as a relationship between:

- the λ-term t
- the LF typing information Θ
- the LF type A
- the LF term M

There are two judgments

$$inv^{\downarrow}(t; A; \Theta) = M \qquad \text{and} \qquad inv^{\uparrow}(t; A; \Theta) = M$$

The first expects A as input while the second synthesizes A.

Returning to our example:

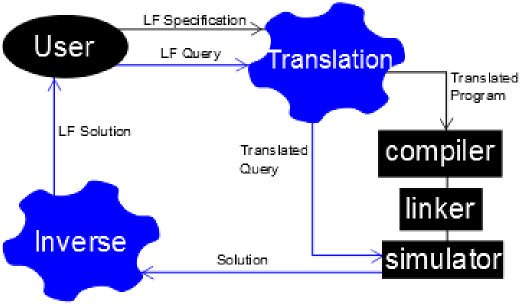
Solution $\quad L = \lambda y.cons\ z\ (cons\ y\ nil)$

$\quad Proof = \lambda y.app\_N\ (cons\ z\ (cons\ y\ nil))$

LF Solution $\quad L = \lambda y{:}nat.cons\ z\ (cons\ y\ nil)$

$\quad Proof = \lambda y{:}nat.app\_N\ (cons\ z\ (cons\ y\ nil))$

Map of Talk


User
LF Specification
LF Query
Translation
Translated Program
LF Solution
Translated Query
compiler
linker
Inverse
Solution
simulator

Ongoing Work

- Show correctness of this translation.
- Developing an implementation of this system.
- Use this translation to extend Abella for reasoning about LF specifications.

End

Correctness of the Translation

- We need to show that the substitutions found in LF and under the translation are 'equivalent'.
- Our approach for this proof is to use simulation.

Theorem

Let Σ be an LF signature and let A be an LF type that possibly contains meta-variables.

1. *If the query $M : A$ is solved with the ground answer substitution σ, then there is an invertible answer substitution θ for the goal $\{\!\{A\}\!\} \langle M \rangle$ wrt $\{\!\{\Sigma\}\!\}$ such that the inverse θ' of θ generalizes σ (i.e. there exists a σ' such that $\sigma' \circ \theta' = \sigma$).*
2. *If θ is an invertible answer substitution for $\{\!\{A\}\!\} \langle M \rangle$, then its inverse is an answer substitution for $M : A$.*

Rules for the Strictness Criterion

$$\frac{dom(\Gamma); \cdot; x \sqsubset_o A_i \text{ for some } A_i \text{ in } \overrightarrow{A}}{\Gamma; x \sqsubset_t c\,\overrightarrow{A}}\ \mathsf{APP_t} \qquad \frac{\Gamma, y : A; x \sqsubset_t B}{\Gamma; x \sqsubset_t \Pi y{:}A.B}\ \mathsf{PI_t}$$

$$\frac{\Gamma_1; x \sqsubset_t B \qquad \Gamma_1, y : B, \Gamma_2; y \sqsubset_t A}{\Gamma_1, y : B, \Gamma_2; x \sqsubset_t A}\ \mathsf{CTX_t}$$

$$\frac{y_i \in \delta \text{ for each } y_i \text{ in } \overrightarrow{y} \qquad \text{each variable in } \overrightarrow{y} \text{ is distinct}}{\Delta; \delta; x \sqsubset_o x\ \overrightarrow{y}}\ \mathsf{INIT_o}$$

$$\frac{y \notin \Delta \text{ and } \Delta; \delta; x \sqsubset_o M_i \text{ for some } M_i \text{ in } \overrightarrow{M}}{\Delta; \delta; x \sqsubset_o y\ \overrightarrow{M}}\ \mathsf{APP_o}$$

$$\frac{\Delta; \delta, y; x \sqsubset_o M}{\Delta; \delta; x \sqsubset_o \lambda y{:}A.M}\ \mathsf{ABS_o}$$

Rules for the Inverse Encoding

$$\frac{X : A \in \Delta}{inv^{\uparrow}(X; A; \Theta) = X}\ \textit{inv-var}$$

$$\frac{inv^{\downarrow}(M; B; \Theta, x : A) = M'}{inv^{\downarrow}(\lambda x.M; \Pi x{:}A.B; \Theta) = \lambda x{:}A.M'}\ \textit{inv-abs}$$

$$\frac{inv^{\uparrow}(M_1; \Pi x{:}B.A; \Theta) = M_1' \qquad inv^{\downarrow}(M_2; B; \Theta) = M_2'}{inv^{\uparrow}(M_1\ M_2; A[M_2'/x]; \Theta) = M_1'\ M_2'}\ \textit{inv-app}$$

$$\frac{u : A \in \Theta}{inv^{\uparrow}(u; A; \Theta) = u}\ \textit{inv-const} \quad \frac{inv^{\uparrow}(M; A; \Theta) = M'}{inv^{\downarrow}(M; A; \Theta) = M'}\ \textit{inv-syn}$$