A Translation-Based Animation of Specifications in a Dependently-Typed Lambda Calculus

Mary Southern and Gopalan Nadathur

Department of Computer Science and Engineering
University of Minnesota

Other Contributors: D. Baelde, Z. Snow

The Context and Objective for this Work

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 that also manipulate objects with binding structure:

- 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 Virtues: 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

Overview of the Talk

We will cover the following topics in this talk

- The dependently typed λ-calculus approach
- The predicate logic approach
- A translation from the first to the second
- Conclusion and future work

Specifications

- Syntax driven, rule based specifications

Lists

$$L := [] \mid (N :: L)$$

Append - $append\ L_1\ L_2\ L_3$

$$\frac{}{append\ []\ L\ L} \qquad \frac{append\ L_1\ L_2\ L_3}{append\ (N :: L_1)\ L_2\ (N :: L_3)}$$

Edinburgh Logical Framework (LF)

Syntax of Expressions

Kind $\quad K := \mathit{Type} \mid \Pi x{:}A.\ K$

Type $\quad A := a \mid \Pi x{:}A.\ B \mid A\ M$

Object $\quad M := c \mid x \mid \lambda x{:}A.M \mid M\ N$

Notation: Write $A \to B$ for $\Pi x{:}A.B$ if x is not free in B

$\mathit{list} : \mathit{Type}$

$\mathit{nil} : \mathit{list}$

$\mathit{cons} : \mathit{nat} \to \mathit{list} \to \mathit{list}$

$\mathit{append} : \mathit{list} \to \mathit{list} \to \mathit{list} \to \mathit{Type}$

$\mathit{appNil} : \Pi L{:}\mathit{list}.\ \mathit{append}\ \mathit{nil}\ L\ L$

$\mathit{appCons} : \Pi N{:}\mathit{nat}.\ \Pi L_1{:}\mathit{list}.\ \Pi L_2{:}\mathit{list}.\ \Pi L_3{:}\mathit{list}.$
$\mathit{append}\ L_1\ L_2\ L_3 \to \mathit{append}\ (\mathit{cons}\ N\ L_1)\ L_2\ (\mathit{cons}\ N\ L_3)$

Signature: A signature identifies constants and provides their types (kinds).

Judgement: $\Gamma \vdash_\Sigma M : A$

Logic Programming with Twelf

Solve judgments of the form $\cdot \vdash_\Sigma M : A$.

- Types as formulas
- Terms as derivations
- Proof search becomes a question of inhabitation

Example

$append : list \to list \to list \to type$
$appNil : append\ nil\ L\ L$
$appCons : append\ L_1\ L_2\ L_3 \to append\ (cons\ X\ L_1)\ L_2\ (cons\ X\ L_3)$

Query: $\cdot \vdash_\Sigma M : append\ (cons\ z\ nil)\ nil\ (cons\ z\ nil)$

Solution: $M = appCons\ z\ nil\ nil\ nil\ (appNil\ nil)$

Specifications Using a Predicate Logic

In this setting, we use the following ideas to encode a formal system

- We identify predicates to represent relevant judgements
- We then use formulas to encode the rules for determining when these judgements hold

Ideally, we would want the formulas also to capture the way the rules can be used to construct derivations in the object system

To realize this requirement, we must restrict the permitted formulas to obtain the desired proof search behavior

A logic with these properties is that of *higher-order hereditary Harrop formulas* that underlies the λProlog language

Higher-Order Hereditary Harrop Formulas (*hohh*)

The atomic formulas in this logic are constructed using predicate symbols that take simply typed λ-terms as arguments

The formulas that are used are then the following

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

A collection of D-formulas encodes a specification and a G formula corresponds to a query

For example, the list specification can be formalized as follows:

$$\begin{array}{lcl} \mathit{list} & : & \mathit{type} \\ \mathit{nil} & : & \mathit{list} \\ \mathit{cons} & : & (\mathit{nat} \rightarrow \mathit{list} \rightarrow \mathit{list}) \end{array}$$

$$\mathit{append} \quad : \quad (\mathit{list} \rightarrow \mathit{list} \rightarrow \mathit{list} \rightarrow o)$$

$$\forall l.(\mathit{append}\ \mathit{nil}\ l\ l)$$
$$\forall x.\forall l_1.\forall l_2.\forall l_3.(\mathit{append}\ l_1\ l_2\ l_3) \supset (\mathit{append}\ (\mathit{cons}\ x\ l_1)\ l_2\ (\mathit{cons}\ x\ l_3))$$

From LF specifications to *hohh* specifications

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 *hohh* terms denoting LF objects and LF types

   In particular,

   - the predicate *hastype* : *lf-obj* $\rightarrow$ *lf-type* $\rightarrow$ *o* is used in this encoding
   - A type itself becomes a formula of one arguement that should be satisfied by any term that has that type

The Translation of LF Types

Let $\langle U \rangle$ denote the simply typed λ-term representing the LF object or type U

Then, LF types are translated as follows:

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

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

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

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

Thus, the LF signature item $\mathit{appNil} : \Pi L{:}\mathit{list}.\ \mathit{append}\ \mathit{nil}\ L\ L$ yields the *hohh* clause

$\forall L.\ (\mathit{hastype}\ L\ \mathit{list}) \supset (\mathit{hastype}\ (\mathit{appNil}\ L)\ (\mathit{append}\ \mathit{nil}\ L\ L)$

A Deficiency with the Translation

We are interested in finding inhabitants of LF types

For example, we might want an M such that the following holds:

$M : (\mathit{append}\ \mathit{nil}\ (\mathit{cons}\ z\ \mathit{nil})\ (\mathit{cons}\ z\ \mathit{nil}))$

- Notice that we already know the type to be well-formed here

In the translation based approach, we would try to solve the query

$\mathit{hastype}\ M\ (\mathit{append}\ \mathit{nil}\ (\mathit{cons}\ z\ \mathit{nil})\ (\mathit{cons}\ z\ \mathit{nil}))$

using the clause

$\forall L.\ (\mathit{hastype}\ L\ \mathit{list}) \supset (\mathit{hastype}\ (\mathit{appNil}\ L)\ (\mathit{append}\ \mathit{nil}\ L\ L)$

However, this requires solving the redundant goal

$\mathit{hastype}\ (\mathit{cons}\ z\ \mathit{nil})\ \mathit{list}$

Can we avoid such redundancies by simplifying the clause to

$\forall L.\ \mathit{hastype}\ (\mathit{appNil}\ L)\ (\mathit{append}\ \mathit{nil}\ L\ L)$?

Improving the Translation

The general question:

When we can use the translation

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

without losing information?

We have identified a sufficient condition for this purpose:

- x must appear in the target type of B
- It must have at least one occurrence which will not disappear after performing substitutions for other quantified variables
- If this occurrence is applied to arguments, they will not change the shape of any term substituted for x

This condition, called strictness, can actually be improved to embody a recursive structure

A Translation-Based Implementation of Twelf

- The approach described here has been tested in the Parinati system that uses Teyjus to execute λProlog programs
- Experiments show that we can get up to an order of magnitude improvement in speed in comparison to an ML implementation
- Space conservation is even more dramatic: really large examples can be run without problem using our approach
- We are now implementing the system in a way that makes Teyjus an opaque backend to Twelf
- This implementation uses an inverse translation from simply typed λ-terms to LF expressions that we have developed

Conclusion

We have developed the theoretical underpinnings for a translation from LF specifications to *hohh* specifications

This translation has benefits at least for animating LF specification

Future Work

- Realize a complete implementation of logic programming in Twelf based on the translation approach.
- Translate totality checking into Abella proofs.
- Reasoning directly using a dependently typed logic in an Abella-like system.