A Translation-Based Approach to Reasoning About Dependently Typed Specifications[1]

Mary Southern

Department of Computer Science and Engineering
University of Minnesota

Joint work with Kaustuv Chaudhuri, INRIA

[1]Based on work which will be presented at FSTTCS 2014

Map of Talk

Motivation

Our Solution

Conclusions

Map of Talk

Motivation

Our Solution

Conclusions

Two-Level Logic Approach 1/2

People are interested in reasoning about formal systems. For example:

- Programs
- Languages
- Logics

To do this we reason about a specification of the system.

- Reasoning in this way is sensible when the specification is an adequate encoding of the system
- Thus properties of the specification are easily translated into properties of the system

Two-Level Logic Approach 1/2

People are interested in reasoning about formal systems. For example:

- Programs
- Languages
- Logics

To do this we reason about a specification of the system.

- Reasoning in this way is sensible when the specification is an adequate encoding of the system
- Thus properties of the specification are easily translated into properties of the system

One approach to this kind of reasoning is called the two-level logic (2LL) approach.

It is characterized by the use of two logics:

1. Specification Logic: for describing the formal system
2. Reasoning Logic: for proving properties about the specification

Where the specification logic can be embedded in the reasoning logic.

Two-Level Logic Approach 2/2

The choice of these logics is closely related.

- Embedding is straightforward when the logics share a type system, and is typically how such systems are designed.
- However, restricting ourselves to using a shared type system does not meet our desire for flexibility in the system

Two-Level Logic Approach 2/2

The choice of these logics is closely related.

- Embedding is straightforward when the logics share a type system, and is typically how such systems are designed.
- However, restricting ourselves to using a shared type system does not meet our desire for flexibility in the system

Two specific drawbacks of this restriction:

1. Modifying the specification logic then requires modification of reasoning logic
2. Only one specification logic can be used in the system

The goal of our work is to provide a method which can relax this restriction.

Abella[2]

Abella is an interactive theorem prover developed by a collaboration between the University of Minnesota and INRIA which uses the two- level logic approach to reasoning.

- The specification logic is that of higher-order herreditary Harrop formulas (hohh), which is a subset of the logic programming language λProlog.
- The reasoning logic of Abella is an intuitionistic predicate logic, $\mathcal{G}$, based on Church's simple theory of types.

Both of these logics are simply typed.

[2] The Abella prover is available from `http://abella-prover.org`

Abella[2]

Abella is an interactive theorem prover developed by a collaboration between the University of Minnesota and INRIA which uses the two- level logic approach to reasoning.

- The specification logic is that of higher-order herreditary Harrop formulas (hohh), which is a subset of the logic programming language λProlog.
- The reasoning logic of Abella is an intuitionistic predicate logic, $\mathcal{G}$, based on Church's simple theory of types.

Both of these logics are simply typed.

- What if we would like to reason about dependently typed specifications in Abella?
  - In a dependedntly-typed logic types may depend on terms
  - This provides an elegant means of encoding relations between terms

  To do this naively would require work on the entire Abella system.

[2] The Abella prover is available from `http://abella-prover.org`

Example - LF specification

- Lists indexed by length.

```
nat : type.
z   : nat.
s   : nat -> nat.

list  : nat -> type.
nil   : list z.
cons  : {N : nat} nat -> list N -> list (s N).
```

Example - LF specification

- Lists indexed by length.

```
nat  : type.
z    : nat.
s    : nat -> nat.

list   : nat -> type.
nil    : list z.
cons   : nat -> list N -> list (s N).
```

Example - LF specification

- Lists indexed by length.

```
nat : type.
z   : nat.
s   : nat -> nat.

list  : nat -> type.
nil   : list z.
cons  : nat -> list N -> list (s N).
```

- Relation on lists.

```
append     : list N1 -> list N2 -> list N3 -> type.
appendNil  : append nil L L.
appendCons : append L1 L2 L3 ->
               append (cons X L1) L2 (cons X L3).
```

Map of Talk

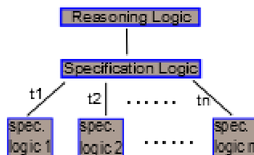
Motivation

Our Solution

Conclusions

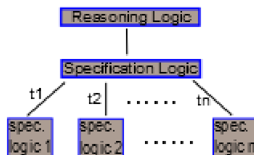
Our Approach 1/2

We propose the use of a translation layer to allow for reasoning about differing specification logics using the same reasoning logic.



Our Approach 1/2

We propose the use of a translation layer to allow for reasoning about differing specification logics using the same reasoning logic.

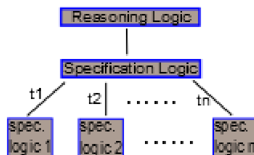


There are two important components to this translation layer:

1. A meaning preserving Translation of the new/modified Specification logic into the original specification logic
2. A Reverse Encoding which can return translated expressions to the new/modified specification logic

Our Approach 1/2

We propose the use of a translation layer to allow for reasoning about differing specification logics using the same reasoning logic.



There are two important components to this translation layer:

1. A meaning preserving Translation of the new/modified Specification logic into the original specification logic
2. A Reverse Encoding which can return translated expressions to the new/modified specification logic

This approach moderates the drawbacks stated previously:

1. We do not need to make modifications to the entire system
2. Multiple specification logics, using suitable translations, may coexist in the system

Our Approach 2/2

Concretely, we have implemented an extension of the Abella prover which can reason about specifications written in the Logical Framework (LF).

The important pieces of achieving this came from previous work:

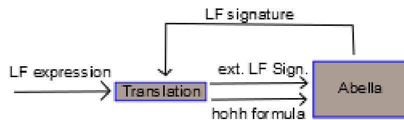
1. The translation we use in our implementation is based on work by Snow, Baelde, and Nadathur
   - Their paper first presents a simple encoding; this is the version of translation we use
   - They also present some optimizations for this basic translation
2. The reverse encoding used by our implementation is based on the inverse relation described in work by Southern and Nadathur.

We will look at each of these pieces in some more detail.

- Note that the solution we used for LF will work for anything which can be adequately encoded into hohh.

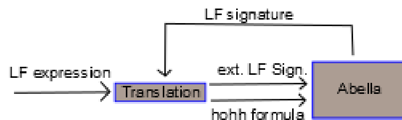
Translation

The translation layer will 'capture' LF input and transform it before passing along to the core of Abella.



Translation

The translation layer will 'capture' LF input and transform it before passing along to the core of Abella.



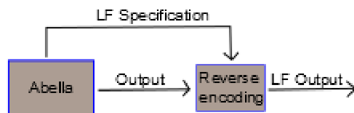
The translation we use for encoding LF into hohh utilizes two phases.

1. First, all LF constants are encoded as hohh constants with types equivalent to the encoding of the flattened LF type.
2. Next, LF judgments are transformed into hohh formulas using the previously generated signature and a 'hastype' predicate.
   - we intend `hastype A B` to mean that `A` is an encoded LF term whose type was encoded as `B`.

Because the signature generated from the first phase is necessary for later translation steps this information is stored by Abella for future use.

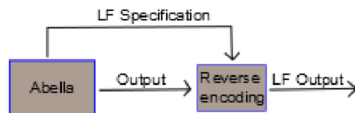
Reversing the Translation

Similar to the translation we 'capture' output from Abella and transform it before printing.



Reversing the Translation

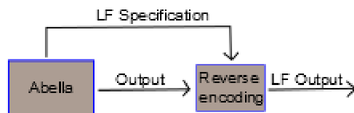
Similar to the translation we 'capture' output from Abella and transform it before printing.



- The key is to use knowledge of LF types

Reversing the Translation

Similar to the translation we 'capture' output from Abella and transform it before printing.



- The key is to use knowledge of LF types

This reverse encoding will keep translation transparent to the user.

- Those familiar with LF may use the system in a natural way, without intimate knowledge of hohh or the translation.
- Clearly separates LF and hohh formulas when working with both logics.

Example - Reasoning

Recall the example specification:

- Lists indexed by length.

```
nat : type.          list  : nat -> type.
z   : nat.           nil   : list z.
s   : nat -> nat.    cons  : nat -> list N -> list (s N).
```

- Relation on lists.

```
append      : list N1 -> list N2 -> list N3 -> type.
appendNil   : append nil L L.
appendCons  : append L1 L2 L3 ->
                 append (cons X L1) L2 (cons X L3).
```

Example - Reasoning

Recall the example specification:

- Lists indexed by length.

```
nat : type.          list  : nat -> type.
z   : nat.           nil   : list z.
s   : nat -> nat.    cons  : nat -> list N -> list (s N).
```

- Relation on lists.

```
append      : list N1 -> list N2 -> list N3 -> type.
appendNil   : append nil L L.
appendCons  : append L1 L2 L3 ->
                 append (cons X L1) L2 (cons X L3).
```

Lets prove that appending lists is unique.

- If `append L1 L2 L3` and `append L1 L2 L3'` then `L3 = L3'`.

Example - Reasoning in Abella

```
============================
 forall L1 L2 L3 L3' M1 M2, <M1:append L1 L2 L3> ->
   <M2:append L1 L2 L3'> -> L3 = L3'

> induction on 1.
```

Example - Reasoning in Abella

```
IH : forall L1 L2 L3 L3’ M1 M2, <M1:append L1 L2 L3>* ->
       <M2:append L1 L2 L3’> -> L3 = L3’
============================
 forall L1 L2 L3 L3’ M1 M2, <M1:append L1 L2 L3>@ ->
   <M2:append L1 L2 L3’> -> L3 = L3’

> intros.
```

Example - Reasoning in Abella

```
IH : forall L1 L2 L3 L3' M1 M2, <M1:append L1 L2 L3>* ->
       <M2:append L1 L2 L3'> -> L3 = L3'
H1 : <M1:append L1 L2 L3>@
H2 : <M2:append L1 L2 L3'>
============================
 L3 = L3'

> case H1.
```

Example - Reasoning in Abella

```
IH : forall L1 L2 L3 L3' M1 M2, <M1:append L1 L2 L3>* ->
       <M2:append L1 L2 L3'> -> L3 = L3'
H2 : <M2:append nil L3 L3'>
H3 : <L3:list N3>*
============================
 L3 = L3'

Subgoal 2 is:
 cons X L4 = L3'

> case H2.
```

Example - Reasoning in Abella

```
IH : forall L1 L2 L3 L3’ M1 M2, <M1:append L1 L2 L3>* ->
       <M2:append L1 L2 L3’> -> L3 = L3’
H3 : <L3’:list N3>*
H4 : <L3’:list N3>
==============================
 L3’ = L3’

Subgoal 2 is:
 cons X L4 = L3’

> search.
```

Example - Reasoning in Abella

```
IH : forall L1 L2 L3 L3' M1 M2, <M1:append L1 L2 L3>* ->
       <M2:append N1 N2 N3 L1 L2 L3'> -> L3 = L3'
H2 : <M2:append (cons X L4) L2 L3'>
H3 : <X:nat>*
H4 : <L4:list N4>*
H5 : <L2:list N2>*
H6 : <L6:list N6>*
H7 : <lf_1:append L4 L2 L6>*
============================
 cons X L6 = L3'

> case H2.
```

Example - Reasoning in Abella

```
IH : forall L1 L2 L3 L3' M1 M2, <M1:append L1 L2 L3>* ->
       <M2:append L1 L2 L3'> -> L3 = L3'
H3 : <X:nat>*
H4 : <L4:list N4>*
H5 : <L2:list N2>*
H6 : <L6:list N6>*
H7 : <lf_1:append L4 L2 L6>*
H8 : <X:nat>
H9 : <L4:list N4>
H10 : <L2:list N2>
H11 : <L8:list N6>
H12 : <lf_2:append L4 L2 L8>
============================
 cons X L6 = cons X L8

> apply IH to H7 H12.
```

Example - Reasoning in Abella

```
IH : forall L1 L2 L3 L3' M1 M2, <M1:append L1 L2 L3>* ->
       <M2:append L1 L2 L3'> -> L3 = L3'
H3 : <X:nat>*
H4 : <L4:list N4>*
H5 : <L2:list N2>*
H6 : <L8:list N6>*
H7 : <lf_1:append L4 L2 L8>*
H8 : <X:nat>
H9 : <L4:list N4>
H10 : <L2:list N2>
H11 : <L8:list N6>
H12 : <lf_2:append L4 L2 L8>
============================
 cons X L8 = cons X L8

> search.
```

Map of Talk

Motivation

Our Solution

Conclusions

Some Comments

- The extended version of Abella is available at `http://abella-prover.org/lf`
- There are also some examples of its use
  - type uniqueness for simply typed λ calculus
  - Isomorphism of De Bruijn and HOAS representations
  - Divergence of Ω
- Can write theorems which are difficult/impossible to state in Twelf
  - Only need to define one HOAS to De Bruijn type family for both directions of the isomorphism
  - Twelf cannot express the coverage property, but this could be done in Abella

Future Work

- Work on this implementation
  - A few rough edges to work out
  - Prepare a more extensive set of examples
  - Compare with other systems for reasoning about dependently typed specifications, such as Beluga and Twelf
- Continuing On
  - Look into using optimizations in the translation.
  - Try out more instances of using our approach

End

Thank You