A Two-Level Logic Approach to Reasoning About Typed Specification Languages

Mary Southern
University of Minnesota

Kaustuv Chaudhuri
INRIA

2014-12-17
FSTTCS

Outline

- Specifications in the LF type theory
- Reasoning *about* specifications
- The Abella/LF branch
- The type encoding problem
- Perspectives

Specifications in the LF type theory

LF type theory

Running example: natural numbers

LF type theory

Running example: natural numbers

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

list : type.
emp : list.
cons : nat -> list -> list.
```

LF type theory

Running example: natural numbers

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

list : type.
emp : list.
cons : nat -> list -> list.
```

```
sum : nat -> nat -> nat -> type.
sum/z : {M:nat} sum z M M.
sum/s : {M:nat} {N:nat} {K:nat}
        sum M N K -> sum (s M) N (s K).
```

LF type theory

Running example: natural numbers

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

list : type.
emp : list.
cons : nat -> list -> list.
```

```
sum : nat -> nat -> nat -> type.
sum/z : {M:nat} sum z M M.
sum/s : {M:nat} {N:nat} {K:nat}
        sum M N K -> sum (s M) N (s K).
```

```
lsum : list -> nat -> type.
lsum/emp : lsum emp z.
lsum/cons : lsum (cons N L) K <-
               lsum L M <- sum M N K.
```

LF: HOAS

Intrinsically encoded simply typed λ-terms

```
ty : type.
i : ty.
arr : ty -> ty -> ty.

tm : ty -> type.
app : tm (arr A B) -> tm A -> tm B.
abs : (tm A -> tm B) -> tm (arr A B).
```

The term $\lambda x.\, \lambda y.\, x\; y\; (\lambda z.\, y)$ is encoded as:

```
abs [x] abs [y] app (app x y) (abs [z] y)
```

```
of : tm A -> ty -> type.
of/app : of (app M N) B <-
           of M (arr A B) <- of N A.
of/abs : of (abs M) (arr A B) <-
           {x} of x A -> of (M x) B.
```

Reasoning *about* specifications

Example: determinacy of `sum`

Say we want to prove:

For every `M,N,K1,K2:nat`,
if `sum M N K1` *and* `sum M N K2`,
then `K1` = `K2`.

Example: determinacy of `sum`

Say we want to prove:

> *For every* **`M,N,K1,K2:nat`**,
> *if* **`sum M N K1`** *and* **`sum M N K2`**,
> *then* **`K1`** $=$ **`K2`**.

```
eq : nat -> nat -> type.
eq/refl : eq N N.
```

Example: determinacy of `sum`

Say we want to prove:

> *For every* `M,N,K1,K2:nat`*,*
> *if* `sum M N K1` *and* `sum M N K2`*,*
> *then* `K1` $=$ `K2`*.*

```
eq : nat -> nat -> type.
eq/refl : eq N N.
```

```
proof : {M:nat} {N:nat} {K1:nat} {K2:nat}
  sum M N K1 -> sum M N K2 -> eq K1 K2 -> type.
```

Example: determinacy of `sum`

Say we want to prove:

> *For every* `M,N,K1,K2:nat`*,*
> *if* `sum M N K1` *and* `sum M N K2`*,*
> *then* `K1` $=$ `K2`*.*

```
eq : nat -> nat -> type.
eq/refl : eq N N.
```

```
proof : {M:nat} {N:nat} {K1:nat} {K2:nat}
  sum M N K1 -> sum M N K2 -> eq K1 K2 -> type.
```

```
proof/z : proof z M M M (sum/z M) (sum/z M) eq/refl.
proof/s : ...
```

Reasoning (meta) logic

Idea: prove this in a different logic where LF typing derivations are inductive structures.

Reasoning (meta) logic

Idea: prove this in a different logic where LF typing derivations are inductive structures.

Let `<M:A>` stand for: the LF judgement `M:A` is valid.

Reasoning (meta) logic

Idea: prove this in a different logic where LF typing derivations are inductive structures.

Let `<M:A>` stand for: the LF judgement `M:A` is valid.

We must be able to show:

$$
\begin{array}{rcll}
\texttt{<N:nat>} & \equiv & & (\texttt{N} = \texttt{z}) \\
 & & \vee & (\exists \texttt{M}.\, \texttt{N} = \texttt{s M} \wedge \texttt{<M:nat>}) \\
\texttt{<\_:sum M N K>} & \equiv & & (\texttt{M} = \texttt{z} \wedge \texttt{N} = \texttt{K}) \\
 & & \vee & (\exists \texttt{M1}, \texttt{K1}.\quad \texttt{M} = \texttt{s M1} \wedge \texttt{K} = \texttt{s K1} \\
 & & & \qquad\qquad \wedge \quad \texttt{<\_:sum M1 N K1>})
\end{array}
$$

Transformed theorem

$$
\begin{aligned}
&\forall \texttt{M}, \texttt{N}, \texttt{K1}, \texttt{K2}.\\
&\quad \big(\texttt{<M:nat>} \wedge \texttt{<N:nat>} \wedge \texttt{<K1:nat>} \wedge \texttt{<K2:nat>}\\
&\qquad \wedge \texttt{<\_:sum M N K1>} \wedge \texttt{<\_:sum M N K2>}\big)\\
&\qquad\quad \supset \texttt{K1} = \texttt{K2}
\end{aligned}
$$

Transformed theorem

$$
\begin{array}{rcll}
\texttt{<N:nat>} & \equiv & & (\texttt{N} = \texttt{z}) \\
& & \vee & (\exists \texttt{M}.\, \texttt{N} = \texttt{s M} \wedge \texttt{<M:nat>}) \\
\texttt{<\_:sum M N K>} & \equiv & & (\texttt{M} = \texttt{z} \wedge \texttt{N} = \texttt{K}) \\
& & \vee & (\exists \texttt{M1}, \texttt{K1}.\quad \texttt{M} = \texttt{s M1} \wedge \texttt{K} = \texttt{s K1} \\
& & & \qquad \wedge \quad \texttt{<\_:sum M1 N K1>})
\end{array}
$$

$$
\begin{array}{l}
\forall M, N, K1, K2. \\
\quad (\texttt{<M:nat>} \wedge \texttt{<N:nat>} \wedge \texttt{<K1:nat>} \wedge \texttt{<K2:nat>} \\
\qquad \wedge \texttt{<\_:sum M N K1>} \wedge \texttt{<\_:sum M N K2>}) \\
\qquad\quad \supset K1 = K2
\end{array}
$$

Transformed theorem

$$
\begin{array}{rcl}
\texttt{<N:nat>} & \equiv_\mu & (\texttt{N} = \texttt{z}) \\
 & \vee & (\exists \texttt{M}.\, \texttt{N} = \texttt{s M} \wedge \texttt{<M:nat>}) \\
\texttt{<\_:sum M N K>} & \equiv_\mu & (\texttt{M} = \texttt{z} \wedge \texttt{N} = \texttt{K}) \\
 & \vee & (\exists \texttt{M1}, \texttt{K1}.\quad \texttt{M} = \texttt{s M1} \wedge \texttt{K} = \texttt{s K1} \\
 & & \qquad\qquad \wedge \quad \texttt{<\_:sum M1 N K1>})
\end{array}
$$

$$
\begin{array}{l}
\forall \textit{M}, \textit{N}, \textit{K1}, \textit{K2}. \\
\quad (\textit{<M:nat>} \wedge \textit{<N:nat>} \wedge \textit{<K1:nat>} \wedge \textit{<K2:nat>} \\
\qquad \wedge\ \textit{<\_:sum M N K1>} \wedge \textit{<\_:sum M N K2>}) \\
\qquad\quad \supset \textit{K1} = \textit{K2}
\end{array}
$$

Abella/LF

Abella

`http://abella-prover.org`

- Interactive tactics-based theorem prover
- Supports:
  - First-order intuitionistic logic
  - Inductive and co-inductive predicate definitions
  - Intensional (structural) equality
  - Generic reasoning and nominal abstraction
- Philosophy:
  - Simple and rigorous proof theory
  - Relational view
  - Pattern unification part of the kernel

Abella/LF

`http://abella-prover.org/lf`

- Ability to "load" arbitrary LF signatures.
- LF typing judgement `<_:_>` is an inductive definition.

Abella/LF Example

nat.elf

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

sum : nat -> nat -> nat -> type.
sum/z : {M:nat} sum z M M.
sum/s : {M:nat} {N:nat} {K:nat}
        sum M N K -> sum (s M) N (s K).
```

Abella/LF Example

nat.thm

```
Specification "nat.elf".

Theorem sum_det3 : forall M N K1 K2,
  <M:nat> -> <N:nat> -> <K1:nat> -> <K2:nat> ->
  (exists P1, <P1:sum M N K1>) ->
  (exists P2, <P2:sum M N K2>) ->
  K1 = K2.
induction on 1.
intros Mnat Nnat K1nat K2nat sum1 sum2.
sum1 : case sum1. sum2 : case sum2.
Mcase : case Mnat. % cases for <M:nat>
 % case of M = z
 sum1 : case sum1. sum2 : case sum2.
   search. % goal was K2 = K2
 % case of M = s M1
 sum1 : case sum1. sum2 : case sum2.
   Knat : case K1nat. K3nat : case K2nat.
   apply IH to Mcase Nnat Knat K3nat _ _.
   search. % goal was s K3 = s K3
```

Abella/LF Example

`nat.thm` (contd.)

```
Theorem lsum_det2 : forall L M1 M2,
  <L:list> -> <M1:nat> -> <M2:nat> ->
  (exists P1, <P1:lsum L M1>) ->
  (exists P2, <P2:lsum L M2>) ->
  M1 = M2.
induction on 1.
intros Llist M1nat M2nat lsum1 lsum2.
lsum1 : case lsum1. lsum2 : case lsum2.
Lcase : case  Llist.
  case lsum1. case lsum2. search.
lsum1c : case lsum1. lsum2c : case lsum2.
  apply IH to Lcase1 lsum1c2 lsum2c2 _ _.
  apply sum_det3 to Lcase lsum1c2 lsum1c3 lsum2c3 _ _.
  search.
```

The type encoding problem

Terms occur at two levels

```
Theorem sum_det3 : forall M N K1 K2,
  <M:nat> -> <N:nat> -> <K1:nat> -> <K2:nat> ->
  (exists P1, <P1:sum M N K1>) ->
  (exists P2, <P2:sum M N K2>) ->
  K1 = K2.
```

Terms occur at two levels

```
Theorem sum_det3 : forall M N K1 K2,
  <M:nat> -> <N:nat> -> <K1:nat> -> <K2:nat> ->
  (exists P1, <P1:sum M N K1>) ->
  (exists P2, <P2:sum M N K2>) ->
  K1 = K2.
```

What are the types of `K1` and `K2` *in the meta-logic*?

Terms occur at two levels

```
Theorem sum_det3 : forall M N K1 K2,
  <M:nat> -> <N:nat> -> <K1:nat> -> <K2:nat> ->
  (exists P1, <P1:sum M N K1>) ->
  (exists P2, <P2:sum M N K2>) ->
  K1 = K2.
```

What are the types of `K1` and `K2` *in the meta-logic*?

- "`nat`": this would mean that Abella/LF's types must at least contain all LF types.

Terms occur at two levels

```
Theorem sum_det3 : forall M N K1 K2,
  <M:nat> -> <N:nat> -> <K1:nat> -> <K2:nat> ->
  (exists P1, <P1:sum M N K1>) ->
  (exists P2, <P2:sum M N K2>) ->
  K1 = K2.
```

What are the types of `K1` and `K2` *in the meta-logic*?

- "`nat`": this would mean that Abella/LF's types must at least contain all LF types.
- "something else": but what?

Why not "`nat`"?

- Type systems are not as canonical as logic:
  - There are zillions of type systems
  - There is only one intuitionistic first-order logic (up to variations in the proof system)
  - Specializing logic to particular type systems therefore seems like a mistake
- Think: two incompatible typing judgements in the same theorem.
  - Happens all the time in translations from one typed language to another
  - Would still like to prove properties of such translations

Un(i)typed encodings

- Insight: assumptions of the form `<M:nat>` already contain all the typing information necessary.

Un(i)typed encodings

- Insight: assumptions of the form **`<M:nat>`** already contain all the typing information necessary.
- So, we can use a single “type” of LF objects, **`lfobj`**.
- Abella’s reasoning logic remains simply typed.

Un(i)typed encodings

- Insight: assumptions of the form **`<M:nat>`** already contain all the typing information necessary.
- So, we can use a single “type” of LF objects, **`lfobj`**.
- Abella’s reasoning logic remains simply typed.

Forgetful type mapping $\phi\,(-)$:

$$\phi\,(A \texttt{ -> } B) \stackrel{\text{def}}{=} \phi\,(A) \to \phi\,(B)$$

$$\phi\,(\texttt{\{}x{:}A\texttt{\}}B) \stackrel{\text{def}}{=} \phi\,(A) \to \phi\,(B)$$

$$\phi\,(\texttt{a}\ M_1\ \cdots\ M_k) \stackrel{\text{def}}{=} \texttt{lfobj}$$

$$\phi\,(\texttt{type}) \stackrel{\text{def}}{=} \texttt{lftype}$$

Translating terms

- To match tye type mapping, we also map dependently typed terms to simply typed terms.

Translating terms

- To match tye type mapping, we also map dependently typed terms to simply typed terms.

Term mapping $\langle - \rangle$:

$$\langle [x{:}A]\, M \rangle \stackrel{\text{def}}{=} \lambda x{:}\phi\,(A).\ \langle M \rangle$$
$$\langle M\ N \rangle \stackrel{\text{def}}{=} \langle M \rangle\ \langle N \rangle$$
$$\langle x \rangle = x$$

Typing as a logic program

- The final ingredient is to recover the typing information that was forgotten in the form of the predicate **`hastype`**:

$$\texttt{hastype} : \texttt{lfobj} \to \texttt{lftype} \to \texttt{o}$$

$$\texttt{istype} : \texttt{lftype} \to \texttt{o}$$

Typing as a logic program

- The final ingredient is to recover the typing information that was forgotten in the form of the predicate **hastype**:

$$\texttt{hastype} : \texttt{lfobj} \to \texttt{lftype} \to \texttt{o}$$

$$\texttt{istype} : \texttt{lftype} \to \texttt{o}$$

Typing program $\{\{-\}\}$:

$$\{\{\{x{:}A\}B\}\} \stackrel{\text{def}}{=} \lambda m{:}(\phi\,(A) \to \phi\,(B)).$$
$$\texttt{pi}\, x{:}\phi\,(A)\,.\,\{\{A\}\}x \;\texttt{=>}\; \{\{B\}\}(m\;x)$$

$$\{\{\texttt{a}\; M_1 \;\cdots\; M_k\}\} \stackrel{\text{def}}{=} \lambda m{:}\texttt{lfobj}.$$
$$\texttt{hastype}\; m\; (\texttt{a}\,\langle M_1\rangle \cdots \langle M_k\rangle)$$

$$\{\{\texttt{type}\}\} \stackrel{\text{def}}{=} \lambda t{:}\texttt{lftype}.\,\texttt{istype}\; t$$

Importing LF signatures

For every signature constant of the form

$$\mathsf{c} : P$$

(where P is a type or a kind):

1. Add this to the meta-signature:

$$\mathsf{c} : \phi\,(P)$$

2. Add a new program clause:

$$\{\{P\}\}\mathsf{c}.$$

In action

```
Abella < Specification "nat.elf".
Reading specification "nat.elf"
sig nat.
  type nat lftype.
  type z lfobj.
  type s lfobj -> lfobj.
  type sum lfobj -> lfobj -> lfobj -> lftype.
  type sum/z lfobj -> lfobj.
  type sum/s lfobj -> lfobj -> lfobj -> lfobj -> lfobj.
end.
module nat.
  (* nat:type *)
  istype nat.
  (* z:nat *)
  hastype z nat.
  (* s:nat -> nat *)
  pi lf_1\ hastype lf_1 nat => hastype (s lf_1) nat.
  (* sum:nat -> nat -> nat -> type *)
  pi lf_1\ hastype lf_1 nat =>
    pi lf_2\ hastype lf_2 nat =>
      pi lf_3\ hastype lf_3 nat => istype (sum lf_1 lf_2 lf_3).
  (* sum/z:{M:nat} sum z M M *)
  pi M\ hastype M nat => hastype (sum/z M) (sum z M M).
  (* sum/s:{M:nat} {N:nat} {K:nat} sum M N K -> sum (s M) N (s K) *)
  pi M\ hastype M nat =>
    pi N\ hastype N nat =>
      pi K\ hastype K nat =>
        pi lf_1\ hastype lf_1 (sum M N K) =>
          hastype (sum/s M N K lf_1) (sum (s M) N (s K)).
end.
```

Adequacy: from Abella back to LF

This is an adequate encoding, meaning that LF typing derivations are in bijection with logic programming derivations using the translated signature.

Adequacy: from Abella back to LF

This is an adequate encoding, meaning that LF typing derivations are in bijection with logic programming derivations using the translated signature.

Every LP derivation can therefore be inverted into an LF typing derivation.

Adequacy: from Abella back to LF

This is an adequate encoding, meaning that LF typing derivations are in bijection with logic programming derivations using the translated signature.

Every LP derivation can therefore be inverted into an LF typing derivation.

Hence, the encoding is invisible for the end user.

Perspectives

Larger Abella/LF examples

http://abella-prover.org/lf

A (growing) number of examples of the use of Abella/LF are available:

- Cut-elimination for focused intuitionistic logic
- Bijection of the HOAS and De Bruijn representations of λ-terms
- Typing-uniqueness for simply typed Church-encoded λ-terms
- Co-inductive reasoning: divergence of Ω, partitioning of untyped λ-terms into normal and diverging terms
- Type-preservation for big-step and small-step evaluation for the pure λ-calculus.

Comparing Abella/LF to Twelf

- Abella/LF proofs are *much* easier to understand and maintain.
- Twelf has a number of trusted algorithmic checks that an inductive type family denotes a theorem. In Abella/LF, theorems are written using standard logical inference, and therefore do not need such checks.
  — Of course, this depends on the correctness of the Abella kernel, which performs unification and checks (co-)inductive proofs.
- Twelf has a very high performance LF type checker, which Abella/LF lacks. (We just use Twelf as our type-checker.)
- Twelf can handle implicit syntax, which Abella/LF lacks. (We use Twelf as our elaborator.)

Comparing Abella/LF to Beluga

`http://complogic.cs.mcgill.edu/beluga`

- Beluga is also a multilevel system, but its meta level is a functional programming language with unrestricted recursion.
- Both levels of Beluga are dependently typed, which makes different tradeoffs:
  - + encoded terms have very precise types, but
  - – many properties are harder to state because of type-interference (*e.g.*, placing the same term in two different contexts)
- Abella has next to no automation in its tactics, so it sometimes requires tedious manual proofs. On the other hand, its trusted codebase is considerably simpler.

Summary

Our results:

- Abella/LF can be used to reason about LF specifications.
- It is both backward and forward compatible with the standard Abella/HH.
- In particular, the reasoning logic $\mathcal{G}$ of Abella is left completely undisturbed.
- Our approach is fairly generic with respect to particular type systems. (The typing derivations must be representable as intuitionistic logic programs.) We have a system in development that can use any functional PTS.

Near future:

- Adding native specification-language type checkers to Abella would reduce a dependency on external tools.
- We next plan to look at intersection and refinement types.