Semantics for ML Polymorphism in Isabelle/HOL

Goal: To formalize the semantics of a typed polymorphic language in the Isabelle theorem prover.

Brian Huffman
Portland State University
RPE Project Presentation, October 12, 2006
(Original Presentation May 14, 2004)
Advisor: John Matthews (Mark Jones)

What is a “Semantics of a typed polymorphic language”?

Semantics: A mapping from expressions to values (meanings)

Functional language: Functions may be passed as arguments

Strongly typed: Compiler statically verifies type correctness

Polymorphism: Functions may be used at multiple types

Implicit typing: Programmer does not have to give types for variables

What does "Formalize in the Isabelle theorem prover" mean?

Theorem prover: A computerized tool to help create and verify mathematical theorems

Isabelle/HOL: A theorem prover using Higher Order Logic, with well developed libraries and automatic proof search

Formal proof: A proof done in a theorem prover

Informal proof: Old-fashioned style (pen and paper) proof

Formalize: To adapt informal proofs for a theorem prover, by adding definitions, identifying assumptions, and filling gaps in reasoning (Not a trivial process!)

Why formalize polymorphism?

Information from polymorphic types can help with

- Equational reasoning (Wadler's Free theorems)
- Automatic compiler optimizations (shortcut fusion)
- Information separation (Haskell state monad)

We would like to use these techniques to reason about security-sensitive and safety-critical programs

- *Formalization would greatly increase confidence in correctness of type-based properties*

Why isn't a paper proof good enough?

Informal proofs of type system soundness are not scalable:

- All examples from the literature use very simplified "core" languages
- Real languages are complex: Proofs have tedious, uninteresting details, and it is easy to miss corner cases

Formal type system proofs *are* scalable:

- Java type system has been formalized in Isabelle (Bali project, `http://isabelle.in.tum.de/Bali`)
- Isabelle can automatically handle many tedious details, and prevents errors and omissions

Core-ML Expressions and Types

A *typing* is an expression paired with a valid type:

- $5 :: Int$, $True :: Bool$
- $even :: Int \rightarrow Bool$
- $(\lambda x.x) :: \alpha \rightarrow \alpha$
- $(\lambda x.\lambda y.y) :: \alpha \rightarrow \beta \rightarrow \beta$
- $(\lambda f.f\ 5) :: (Int \rightarrow \alpha) \rightarrow \alpha$

Some expressions have no valid type:

- $even\ True :: error$
- $(\lambda f.f\ f) :: error$

Beta Reduction

Reducible expression (Redex): A function (lambda abstraction) applied to an argument: $(\lambda x.\lambda y.y + x)\ (5)$

Beta reduction: To remove redexes by substitution of an argument into a function body:
$(\lambda x.\lambda y.y + x)\ (5)\ (3) \rightarrow_{\beta} (\lambda y.y + 5)\ (3) \rightarrow_{\beta} (3 + 5)$

Beta normal form: No more beta reductions are possible.

Subject reduction property: Beta reduction preserves types.

Previous Approach to Semantics: Milner-Style

Domain of values V includes subsets V_{Int}, V_{Bool}, and its own function space $[V \to V]$.

Types are subsets of V. Polymorphic types are intersections of their instance types. Type soundness means $e :: t$ implies $[\![e]\!] \in V_t$.

Meaning function is untyped: meanings based only on expression syntax

- $[\![\lambda x.x]\!] =$ the ϕ such that $\phi(v) = v$ for all $v \in V$.
- $[\![\lambda f.f\ f]\!] =$ the ψ such that $\psi(v) = v(v)$ for all $v \in V$.

The value $\phi \in V_{\alpha \to \alpha}$, but ψ is not a member of any type.

Another Possibility: Simply Typed Lambda Calculus ($T\Lambda$)

Bound variables are labeled with simple types (no polymorphism).

- $(\lambda(x{:}\, Bool \to Bool).\lambda(y{:}\, Int).y) :: (Bool \to Bool) \to Int \to Int$

Erasing type labels always yields a valid Core-ML typing:

- $(\lambda x.\lambda y.y) :: (Bool \to Bool) \to Int \to Int$

There exists a $T\Lambda$ typing for every valid Core-ML typing.

Can we define ML semantics in terms of $T\Lambda$ semantics?

Problem: Correspondence is not 1-to-1

Multiple $T\Lambda$ typings can have the same type-erasure.

$$\begin{tikzcd}[column sep=small]
& & (\lambda x.\lambda y.y)(\lambda z.z) :: a \to a \\
(\lambda(x : b \to b).\lambda(y : a).y)(\lambda(z : b).z) :: a \to a \arrow[urr, dashed, "erase"] & & \\
& (\lambda(x : c \to c).\lambda(y : a).y)(\lambda(z : c).z) :: a \to a \arrow[uur, dashed, "erase"'] &
\end{tikzcd}$$

If multiple $T\Lambda$ terms are possible, then which one should we choose?

Solution: Ohori-Style Semantics

Ohori proves that $T\Lambda$ typings with the same type-erasure must all β-reduce to the same value.

$$\begin{tikzcd}
& & (\lambda x.\lambda y.y)(\lambda z.z) :: a \to a \\
(\lambda(x : b \to b).\lambda(y : a).y)(\lambda(z : b).z) :: a \to a \arrow[urr, dashed, "erase"] \arrow[dd, squiggly, "reduce"'] & & \\
& (\lambda(x : c \to c).\lambda(y : a).y)(\lambda(z : c).z) :: a \to a \arrow[uur, dashed, "erase"'] \arrow[dl, squiggly, "reduce"] & \\
(\lambda(y : a).y) :: a \to a & &
\end{tikzcd}$$

By using a semantics of $T\Lambda$ that preserves meaning over β-reduction, it does not matter which $T\Lambda$ typing we choose.

Formalization in Isabelle/HOL

The project includes many sub-theories:

- Types and type environments
- Expressions and beta reduction (Core-ML and $T\Lambda$)
- Typing rules and subject reduction (Core-ML and $T\Lambda$)
- Models of $T\Lambda$
- Normalization properties of $T\Lambda$
- Definitions and proofs of Ohori's main theorems

The remainder of this talk will cover the formalization of Ohori's Theorem 6.

Weak Normalization Property of $T\Lambda$

Weak Normalization Property: For all terms, there exists a reduction path that terminates.

Constructive proof: We define the `reduce` function in Isabelle, and prove these properties:

consts `reduce :: TL` $\Rightarrow$ `TL`

lemma `reduce_rbeta: e` $\twoheadrightarrow_\beta$ `reduce e`

lemma `reduce_beta_normal:`

`e :: t` $\Longrightarrow$ `reduce e` $\in$ `beta_normal`

The existence of this function means that $T\Lambda$ has the Weak Normalization property.

Type Erasure

We formally define type erasure using primitive recursion.

```
consts erasure :: TL ⇒ expr
primrec
  erasure (TL.Var x) = expr.Var x
  erasure (TL.Abs t e) = expr.Abs (erasure e)
  erasure (TL.App u v) =
    expr.App (erasure u) (erasure v)
```

It is easy to show by induction that `erasure` preserves typings and the beta relation.

Confluence of Beta Reduction

Isabelle's libraries prove confluence of beta reduction for untyped lambda calculus.

lemma `confluent_beta:`

$$
\begin{tikzcd}
 & a \arrow[dl, rightsquigarrow, "\beta^*"'] \arrow[dr, rightsquigarrow, "\beta^*"] & \\
b & & c
\end{tikzcd}
\quad \Longrightarrow \quad
\begin{tikzcd}
 & a \arrow[dl, rightsquigarrow, "\beta^*"'] \arrow[dr, rightsquigarrow, "\beta^*"] & \\
b \arrow[dr, rightsquigarrow, "\beta^*"'] & & c \arrow[dl, rightsquigarrow, "\beta^*"] \\
 & d &
\end{tikzcd}
$$

If b and c are both in beta normal form, then $b = c$. (Why?)

Lemma for Theorem 6

For Core-ML terms in beta normal form, there is a 1-1 correspondence with $T\Lambda$:

lemma `beta_normal_erasure_eq:`

$\llbracket$ `e` $\in$ `beta_normal; e :: t; e' :: t;`

`erasure e = erasure e'` $\rrbracket \Longrightarrow$ `e = e'`

For an inductive proof, we need to strengthen the hypothesis:

$\llbracket$ `e` $\in$ `var_app; e :: t; e' :: t';`

`erasure e = erasure e'` $\rrbracket \Longrightarrow$ `e = e'` $\wedge$ `t = t'`

We prove both hypotheses simultaneously by induction.

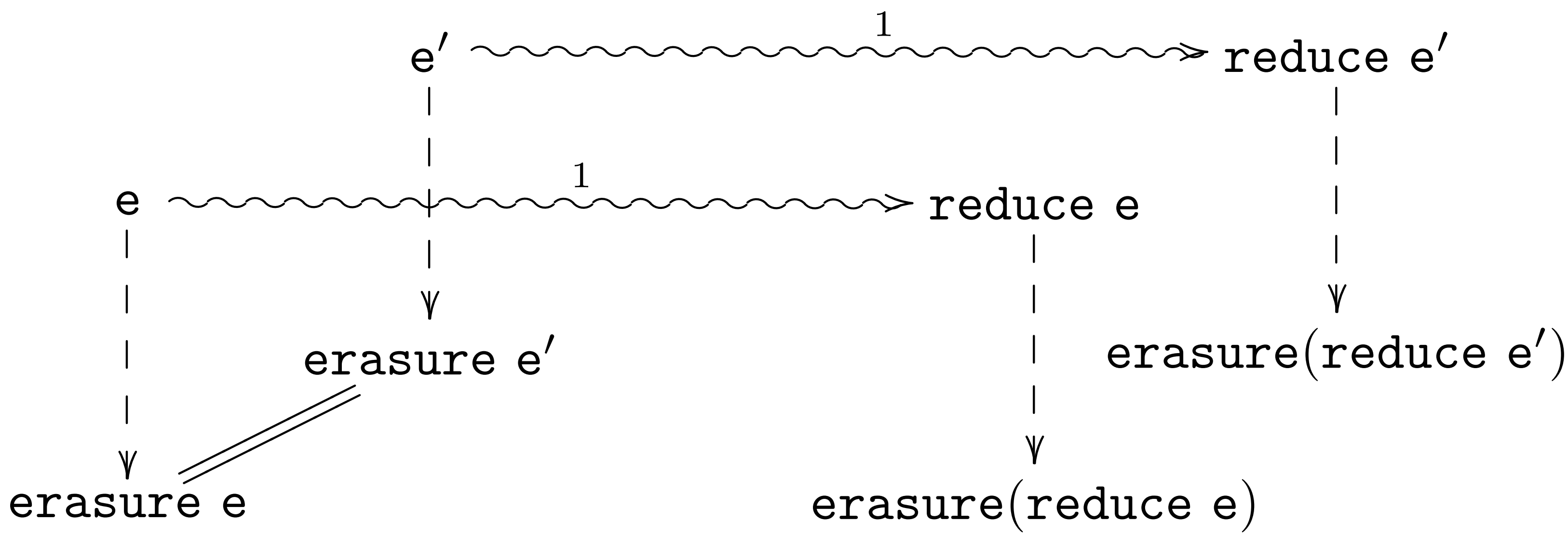
Proof of Theorem 6

This is the central theorem of the Ohori-style semantics:

theorem `Theorem6:`
$\llbracket$ `e :: t; e' :: t; erasure e = erasure e'` $\rrbracket$
$\Longrightarrow$ `reduce e = reduce e'`

The proof uses lemmas from many supporting theories: Weak normalization, subject reduction, confluence, type erasure, and beta normal terms.

Proof of Theorem 6, Continued



- Have `e` $\twoheadrightarrow_\beta$ `reduce e` and `reduce e` $\in$ `beta_normal` by `reduce_rbeta` and `reduce_beta_normal`

Proof of Theorem 6, Continued

$$
\begin{tikzcd}
 & \mathtt{e'} \arrow[rr, squiggly, "1"] \arrow[dd, dashed] & & \mathtt{reduce\ e'} \arrow[dd, dashed] \\
\mathtt{e} \arrow[rr, squiggly, "1"] \arrow[dd, dashed] & & \mathtt{reduce\ e} \arrow[dd, dashed] & \\
 & \mathtt{erasure\ e'} \arrow[rr, squiggly, "2"] \arrow[dl, equal] & & \mathtt{erasure}(\mathtt{reduce\ e'}) \\
\mathtt{erasure\ e} \arrow[rr, squiggly, "2"] & & \mathtt{erasure}(\mathtt{reduce\ e}) &
\end{tikzcd}
$$

- Then have `erasure(e)` $\twoheadrightarrow_\beta$ `erasure(reduce e)` and `erasure(reduce e)` $\in$ `beta_normal`

Proof of Theorem 6, Continued

$$\begin{tikzcd}
 & \texttt{e}' \arrow[rr, rightsquigarrow, "1"] \arrow[dd, dashed] & & \texttt{reduce e}' \arrow[dd, dashed] \\
\texttt{e} \arrow[rr, rightsquigarrow, "1"] \arrow[dd, dashed] & & \texttt{reduce e} \arrow[dd, dashed] & \\
 & \texttt{erasure e}' \arrow[rr, rightsquigarrow, "2"] \arrow[dl, equal] & & \texttt{erasure}(\texttt{reduce e}') \arrow[dl, equal, "3"'] \\
\texttt{erasure e} \arrow[rr, rightsquigarrow, "2"] & & \texttt{erasure}(\texttt{reduce e}) &
\end{tikzcd}$$

- Then have `erasure (reduce e) = erasure (reduce e')` by `confluent_beta`

Proof of Theorem 6, Concluded

$$
\begin{tikzcd}
 & \texttt{e}' \arrow[rr, squiggly, "1"] \arrow[dd, dashed] & & \texttt{reduce e}' \arrow[dl, equal, "4"'] \arrow[dd, dashed] \\
\texttt{e} \arrow[rr, squiggly, "1"] \arrow[dd, dashed] & & \texttt{reduce e} \arrow[dd, dashed] & \\
 & \texttt{erasure e}' \arrow[rr, squiggly, "2"] \arrow[dl, equal] & & \texttt{erasure}(\texttt{reduce e}') \arrow[dl, equal, "3"'] \\
\texttt{erasure e} \arrow[rr, squiggly, "2"] & & \texttt{erasure}(\texttt{reduce e}) &
\end{tikzcd}
$$

- Finally have `reduce e = reduce e'` by `beta_normal_erasure_eq`. Done!

Meaning function

The framework ends by defining the meaning function:

$\textbf{types}\ \alpha\ \texttt{model = [TL}\ \times\ \texttt{typ,}\ \alpha\ \texttt{env]}\ \Rightarrow\ \alpha$

$\textbf{consts}\ \texttt{meaning ::}$

$\quad \texttt{[}\alpha\ \texttt{model,}\ \alpha\ \texttt{env, expr]}\ \Rightarrow\ \texttt{(typ}\ \times\ \alpha\texttt{) set}$

Using Theorem 6, we can prove that the meanings are uniquely defined:

$\textbf{theorem}\ \texttt{single\_valued\_meaning:}$

$\quad \texttt{model M}\ \Longrightarrow\ \texttt{single\_valued (meaning M v e)}$

Using the meaning function, it is now possible to reason formally about Core-ML programs.

Future Directions

Short term goals:

- Build models of $T\Lambda$ in Isabelle
- Prove full-abstraction result for Ohori-style semantics

Long term goal is to add support for more language features:

- General recursion
- Algebraic datatypes (lists and pairs)
- Type classes and overloading

Conclusion

We have formalized a semantics for the core of a polymorphic functional language in the Isabelle theorem prover.

Using the Isabelle theorem prover means that all results meet a very high standard of mathematical rigor.

The formalization includes a body of supporting theories that will be reusable for future work.

- Formalization defines ~25 constants and ~80 lemmas and theorems.
- For comparison: Ohori's paper lists ~10 theories and lemmas.

Thank You

Atsushi Ohori. A Simple Semantics for ML Polymorphism. Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture, November 1990.