Semantics for ML Polymorphism in Isabelle/HOL

Brian Huffman

24th May 2004

Abstract

A framework for an Ohori-style denotational semantics of a polymorphic language is formalized in Isabelle/HOL, an implementation of Higher-Order Logic in the Isabelle theorem prover. The formalization provides a non-trivial case study of many advanced features of Isabelle, including mutually-defined inductive sets, well-founded recursion, and proof locales.

As part of the framework, a new proof of the Weak Normalization property for the simply typed lambda calculus is presented, which uses Isabelle's "recdef" facility for recursive definitions. A function that reduces terms to beta normal form is defined and then proven correct, yielding a completely formalized, concise proof of Weak Normalization.

1 Introduction

Type systems can be useful to programmers for various reasons. The most immediate benefit is that in a type-checked program, run-time type errors are guaranteed not to occur; but the usefulness does not stop there. The types of functions in a program can provide a lot of information about their properties and behavior.

In languages with parametric polymorphism, you can derive theorems about a polymorphic function based solely on its type (termed "free theorems" by Wadler [4]). The intuition is that a polymorphic function can only do so much with its arguments: It can pass them to other functions, or put them in data structures, but it cannot poke around inside them. This kind of reasoning has been used to justify automatic compiler optimizations, like foldr-build fusion and automatic deforestation [9].

Another example, relevant to information separation properties, is Haskell's ST monad: You can encapsulate blocks of imperative code inside the monad, and the type system guarantees that the side effects of such code cannot affect the execution of the rest of the program [10].

As typed programming languages start being used for security-sensitive and safety-critical applications, we will place more and more reliance on the correctness of properties asserted by type systems. For this reason, it is very important for type systems to have a well-defined semantics.

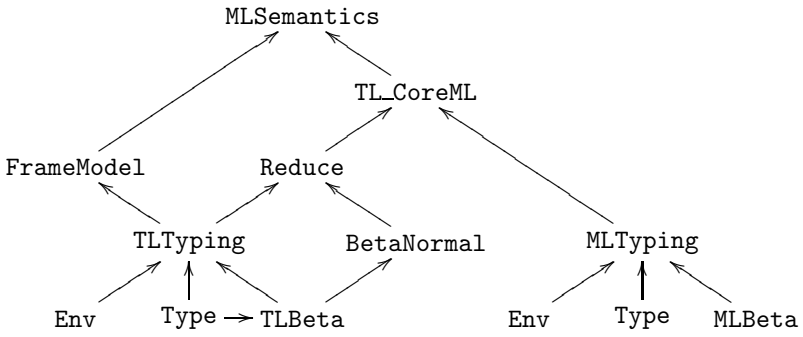

Figure 1: Module dependencies. An arrow from `A` to `B` means that module `B` imports the definitions and theorems from module `A`.

This project formalizes the polymorphic type system of a very simple functional language. It may be considered as a first step towards formalizing a more complicated type system, such as that of the functional language Haskell. A formalized type system semantics would be particularly valuable for Haskell because there has been a good deal of research on equational reasoning with Haskell programs [9, 10, 5], and formalization could greatly increase the level of confidence in those results.

We chose to use Ohori-style semantics because it accurately models the Hindley-Milner polymorphism that underlies languages such as ML and Haskell. Furthermore, it promises to be extensible: The set-of-pairs denotations should adapt to accommodate ad-hoc polymorphism, such as with Haskell type classes.

The framework presented in this paper is divided among several interrelated files; Figure 1 gives a graphical representation of some of the module dependencies. After presenting some background material in Section 2, we then trace the development of the various theory modules, from the bottom up. Section 3 covers the `Type`, `TLBeta`, `MLBeta`, and `BetaNormal` modules, which define expressions and the basic operations on them, including beta reduction. Section 4 discusses the `Env`, `TLTyping`, and `MLTyping` modules, which define typing rules. Section 5 describes the `Reduce`, `FrameModel`, `TL_CoreML`, and `MLSemantics` modules, which complete the formal definition of Ohori-style semantics. The proof of Weak Normalization for the simply typed lambda calculus can be found in the appendix.

2 Background

2.1 Types and Polymorphism

In a typed programming language, every value belongs to a type. Types come in several varieties: *Base types* like `nat` and `bool` represent atomic values, including

numbers and the constants `True` and `False`. *Simple types* are built up from base types using type constructors like `list` and the function arrow ($\rightarrow$). Other names for simple types include *ground types* and *monomorphic types*.

Polymorphic types represent values that can fit into more than one simple type. For example, consider an identity function that maps every input to itself. Depending on the type of the input, the function could be classified as `bool` $\rightarrow$ `bool`, `nat` $\rightarrow$ `nat`, or any other function type of this form. Thus we write $\forall\alpha.\alpha \rightarrow \alpha$ as its polymorphic type. The simple types `bool` $\rightarrow$ `bool`, `nat` $\rightarrow$ `nat`, etc. are called *instances* of $\forall\alpha.\alpha \rightarrow \alpha$.

Hindley-Milner types are polymorphic types where all universal quantifiers are positioned at the outside. Hindley-Milner types are important for implicitly-typed functional languages because they can be automatically generated by type inference algorithms. With Hindley-Milner types, we usually omit writing quantifiers and implicitly quantify over all variables in a type. For example $\alpha \rightarrow \beta \rightarrow \alpha$ stands for $\forall\alpha.\forall\beta.\alpha \rightarrow \beta \rightarrow \alpha$. Hindley-Milner polymorphic types are also known as *type schemes*.

2.2 Ohori-Style Semantics

At the core of most typed functional programming languages, there is a facility for defining polymorphic values. This basic functionality is embodied in Core-ML, an implicitly typed functional language with polymorphic let bindings.

The denotational semantics framework developed by Ohori [1] shows how to build a semantics for Core-ML from any given model $\mathcal{M}$ of simply typed lambda calculus. In Ohori's framework, the meaning of a Core-ML term is a set of pairs, each consisting a simple type together with an element of that type from $\mathcal{M}$; the set contains one entry for every instance of the term's polymorphic type. One important benefit of Ohori-style semantics is that no additional mathematical machinery is needed, beyond what is required to model the simply typed lambda calculus. Another benefit is that meanings are only assigned to typable terms.

Ohori's framework is based on the observation that there is a one-to-one correspondence between Core-ML typing derivations and well-formed terms in the simply typed lambda calculus. This suggests that maybe we could use this correspondence to give meanings to Core-ML terms. But in general, typing derivations for Core-ML terms are not unique; in fact, a single term may have an infinite number of distinct derivations.

The central theorem of Ohori's framework proves this important fact: Simply typed lambda terms that correspond to derivations of the same Core-ML term must be beta-convertible. Therefore, if we use a model of simply typed lambda calculus that gives the same value to beta-convertible terms, then the choice of typing derivation does not matter; each valid Core-ML typing judgment maps to a single value from the model.

2.3 Isabelle/HOL

Isabelle is a generic interactive theorem prover, which can be instantiated with various kinds of object-logics. Isabelle/HOL is an instantiation of Higher Order Logic; it includes many useful features to facilitate definitions, along with a well developed library of pre-existing theories. From now on, I will just use the name Isabelle to refer to Isabelle/HOL.

Isabelle theory files contain definitions, theorems, and proof scripts. In this paper I will often indicate the complexity of proofs, and the key lemmas used in each proof, but I present no actual proof scripts. When I say that a proof is automatic, it means that one of Isabelle's built-in tactics was able to prove it in one step. Other proofs require explicit application of particular lemmas. A common form of proof is to manually apply a specific induction rule, and then finish the proof automatically. For more information on writing Isabelle proof scripts, see the Isabelle Tutorial [11].

The formula syntax in Isabelle includes standard logical notation for connectives and quantifiers. In addition, Isabelle has separate syntax for the meta-level logic: $\bigwedge$, $\Longrightarrow$, and $\equiv$ represent meta-level universal quantification, implication, and equality. There is also notation for nested meta-level implication: $[\![P_1; P_2; \ldots; P_n]\!] \Longrightarrow R$ is short for $P_1 \Longrightarrow P_2 \Longrightarrow \cdots \Longrightarrow P_n \Longrightarrow R$.

The syntax of types is similar to the ML language, except that Isabelle uses a double arrow ($\Rightarrow$) for function types. The pair type constructor ($\times$) is infix, and other type constructors are written postfix: For example, `(nat` $\times$ `bool) list` represents lists of pairs of natural numbers and booleans. In the type syntax, $[t_1, t_2, \ldots, t_n] \Rightarrow r$ is special notation for the function type $t_1 \Rightarrow t_2 \Rightarrow \cdots \Rightarrow t_n \Rightarrow r$.

Isabelle theories declare new constants with the **consts** keyword. Definitions for constants can be given in a variety of ways: **primrec** is for primitive-recursive functions, **recdef** defines functions using well-founded recursion, **inductive** is for inductively-defined sets, and **defs** is for ordinary (non-recursive) definitions. The keywords **lemma** and **theorem** introduce theorems, and **datatype** defines new algebraic datatypes. Other syntax will be discussed later as it is used.

3 Expressions and Substitution

3.1 Datatypes

The framework starts with definitions of basic expressions and types. The syntax of the Core-ML language is just untyped lambda calculus, extended with local definitions:

$$e := x \mid e_1\ e_2 \mid \lambda x.e \mid \text{let } x = e_1 \text{ in } e_2$$

The Isabelle 2003 distribution already comes with a formalization of type inference for Mini-ML, a language syntactically equivalent to Core-ML, which is described in a paper by Naraschewski and Nipkow [3]. To allow my framework to be integrated with existing libraries, I have used their datatype definitions

for expressions and types. We represent Core-ML expressions in de Bruijn notation, using a datatype with constructors for variables, lambda abstractions, applications, and let bindings.

```
datatype expr =
  Var nat | Abs expr | App expr expr | LET expr expr
```

In de Bruijn notation [7], bound variables are implicitly named with numbers, so the variable names on lambda abstractions and let bindings disappear: `Abs e` represents $\lambda x.e[x/0]$, and `LET e1 e2` represents let $x = e_1$ in $e_2[x/0]$. Variables are referenced positionally, so that `Var n` in a subterm means that you must pass `n` abstractions or let bindings, moving outward, before you get to the place where the referenced variable is bound. If `n` is larger than the number of enclosing abstractions, then it references a free variable. example, we represent the expression $\lambda x.(\lambda y.y\ x)$ as `Abs (Abs (Var 0) (Var 1))`, and let $x = (\lambda y.y)$ in $x\ x$ as `LET (Abs (Var 0)) (App (Var 0) (Var 0))`.

The `Type` module declares the datatype of types, which are divided into function types and atomic types. The interpretation of atomic types is not important for now; we can think of them simply as uninterpreted type constants.

```
datatype typ = TVar nat | typ → typ
```

In addition to Core-ML, my framework also uses expressions of simply typed lambda calculus; Ohori's paper refers to this language as $T\Lambda$, which is the reason for the name `TL`. The datatype definition is similar to the one for Core-ML, except that the `Abs` constructor takes an additional type argument: `Abs t e` represents $\lambda x : \tau.e[x/0]$.

```
datatype TL = Var nat | App TL TL | Abs typ TL
```

Note that the `TL` datatype uses the same constructor names as the `expr` datatype: Usually only one or the other will be in scope at a time, but we can resolve any ambiguity by using qualified names, like `TL.App`.

There are advantages and disadvantages to using de Bruijn indices instead of variable names. One disadvantage is that de Bruijn terms are just hard for people to read; another is that they do not accurately model real programming languages, which use variable names. However, de Bruijn notation makes formalization much easier, for several reasons: Firstly, most it is desirable to identify expressions up to renaming of bound variables. Secondly, substitution is much easier because bound variable capture is not a problem. Finally, and perhaps most importantly for this project, there is a already good library support in Isabelle for de Bruijn terms.

3.2 Beta reduction

Included in the Isabelle 2003 distribution is a theory of beta reduction for the untyped lambda calculus, described in a paper by Nipkow [2]. In its module `Lambda`, it defines a datatype for de Bruijn-style lambda terms, as well as substitution and a beta reduction relation.

```
datatype dB = Var nat | App dB dB | Abs dB
consts
  subst :: [dB, dB, nat] ⇒ dB (_[_/_])
  beta :: (dB × dB) set
translations
  x →β y ⇌ (x,y) ∈ beta
  x →→β y ⇌ (x,y) ∈ beta*
inductive beta intros
  beta: App (Abs s) t →β s[t/0]
  appL: s →β t ⟹ App s u →β App t u
  appR: s →β t ⟹ App u s →β App u t
  abs:  s →β t ⟹ Abs s →β Abs t
```

The above declarations define some convenient syntax: Instead of `subst e1 e2 x`, we can use the more traditional notation `e1[e2/x]`, and the **translations** command gives us infix symbols for single-step and multiple-step beta reduction.

The library also provides a well chosen set of lemmas about substitution, which I will not list here. My `MLBeta` and `TLBeta` modules are versions of the `Lambda` module that have been adapted to work with the `expr` and `TL` types, respectively. By re-using this library code, we get a full theory of substitution and beta reduction on each datatype with very little effort.

3.3 Beta normal terms

A lambda term is in *beta normal form* if and only if no further beta reductions of the term are possible. In Isabelle, we can formalize what it means for a term `x` to be in normal form with respect to a reduction relation `r`:

```
consts normal :: [(α × α), α] ⇒ bool
defs normal r x ≡ ∀y. (x,y) ∉ r
```

With this definition, we can use the predicate `normal beta` to test whether a given term is in beta normal form. However, this is a cumbersome definition when it comes to proving theorems about beta normal terms; we would prefer a definition that allows straightforward inductive-style proofs.

We can use Isabelle's inductive set definition facility to make an inductive characterization of beta normal terms; we just need to supply an appropriate set of introduction rules. For example, any single variable is in beta normal form, and a lambda abstraction is in normal form if its body is in normal form. For an application, both of its arguments must be in normal form, and additionally the left argument must not be a lambda abstraction. Because of this extra requirement, our inductive definition must actually distinguish two sets: the set of all beta normal terms, and those beta normal terms that are either variables or applications. Fortunately, Isabelle allows us to define both sets simultaneously.

```
consts
  beta_normal :: TL set
```

```
  var_app     :: TL set
inductive beta_normal var_app intros
  Var x ∈ var_app
  v ∈ var_app ⟹ v ∈ beta_normal
  b ∈ beta_normal ⟹ Abs t b ∈ beta_normal
  ⟦ v ∈ var_app; b ∈ beta_normal ⟧
    ⟹ App v b ∈ var_app
```

For every inductively defined set, Isabelle automatically generates an induction rule for proving properties about members of the set. When two sets are defined simultaneously with mutual induction, we get a rule for simultaneously proving separate properties on each set. Isabelle produces the following rule for our definition of `beta_normal` and `var_app`:

```
beta_normal_var_app.induct:
  ⟦ ⋀v. ⟦ v ∈ var_app; P2 v ⟧ ⟹ P1 v;
    ⋀b t. ⟦ b ∈ beta_normal; P1 b ⟧ ⟹ P1 (Abs t b);
    ⋀x.  P2 (Var x);
    ⋀b v. ⟦ v ∈ var_app; P2 v; b ∈ beta_normal; P1 b ⟧
      ⟹ P2 (App v b)
  ⟧ ⟹ (xb ∈ beta_normal ⟶ P1 xb) ∧
       (xa ∈ var_app ⟶ P2 xa)
```

This induction rule means that if we want to inductively prove some property `P1` of beta normal terms, we can use a stronger inductive hypothesis `P2` for the `var_app` subterms.

Finally, we need to prove that the inductive characterization matches the original definition of beta normal terms.

```
lemma beta_normal_correct:
  (x ∈ beta_normal) = (normal beta x)
```

The correctness proof is automatic, by induction on `x`.

4 Typings

4.1 Type environments

A type environment is a (typically finite) mapping from variables to types, or in the case of lambda terms with de Bruijn indices, a mapping from naturals to types. The standard operations on type environments include looking up types of existing variables, and adding new bindings for fresh variables. The theory module `Env` represents type environments as partial functions, that is, functions that return an optional result.

```
types α env = nat ⇒ α option
consts
```

```
  mapsto :: [α env, nat, α] ⇒ bool
  shift :: [α env, nat, α] ⇒ α env (_{_:=_})
defs
  mapsto A k ≡ (λx.  A k = Some x)
  A{k:=s} ≡ (λi. if k < i then A (i - 1)
     else if i = k then Some s else A i)
```

To make the rest of the framework independent of the particular implementation of type environments, I have taken a few steps to hide the implementation details. The **types** command defines the `env` type constructor as a type synonym, for which the `mapsto` and `shift` functions make up an abstract interface. By supplying a few lemmas, the rest of the framework can get by without using the definitions of `mapsto` and `shift`.

```
lemma mapsto_same:
  ⟦ mapsto A k s; mapsto A k s' ⟧ ⟹ s' = s
lemma mapsto_shift_eq:
  mapsto (A{k:=s}) k s' = (s = s')
lemma mapsto_shift_lt:
  k < x ⟹ mapsto (A{k:=s}) x = mapsto A (x - 1)
lemma mapsto_shift_gt:
  x < k ⟹ mapsto (A{k:=s}) x = mapsto A x
lemma shift_shift:
  A{k:=t1}{0:=t2} = A{0:=t2}{k+1:=t1}
```

All of the above lemmas are easily proved from the definitions of `mapsto` and `shift`.

Partial functions are not the only way to represent type environments in Isabelle, and indeed different libraries within the Isabelle distribution use different representations. The Mini-ML library uses lists of types, and the files included with the Church-Rosser theory use functions from naturals to types. To enable integration with other libraries, I created an alternative `Env` module that uses lists instead of partial functions. In the list implementation, I had to replace `(s = s')` with `(s = s' ∧ k ≤ length A)` in the `mapsto_shift_eq` lemma, but otherwise the alternative module worked perfectly as a drop-in replacement.

4.2 Typing rules

The `MLTyping` and `TLTyping` modules define the typing rules for Core-ML and simply typed lambda calculus, respectively. The typing introduction rules for Core-ML are given below.

```
consts typing :: (typ env × expr × typ) set
translations A ⊢ e : t ⇌ (A,e,t) ∈ typing
inductive typing intros
  var: mapsto A x t ⟹ A ⊢ Var x : t
  abs: A{0:=t1} ⊢ e : t2 ⟹ A ⊢ Abs e : t1 → t2
```

```
app: ⟦ A ⊢ e1 : t1 → t2; A ⊢ e2 : t1 ⟧
     ⟹ A ⊢ App e1 e2 : e2
LET: ⟦ A ⊢ e2[e1/0] : t; A ⊢ e1 : t' ⟧
     ⟹ A ⊢ LET e1 e2 : t
```

The typing rules for the simply typed lambda calculus are similar, except that the `abs` rule includes a type label, and there is no `LET` rule.

After defining the typing rules, each of these theory modules proves a few theorems about how substitution and beta reduction interact with typing.

```
lemma subst_typing:
  ⟦ A{k:=t'} ⊢ e : t; A ⊢ e' : t' ⟧ ⟹ A ⊢ e[e'/k] : t
lemma beta_typing:
  ⟦ e1 →β e2; A ⊢ e1 : t ⟧ ⟹ A ⊢ e2 : t
```

The `subst_typing` lemma is proved by induction over `e`, with some manual assistance. Using `subst_typing` as a lemma, `beta_typing` is then proved automatically by induction over the beta relation. The `beta_typing` theorem shows that these typing rules have the subect reduction property. As a corollary to `beta_typing`, it is easy to show that the transitive closure of the beta relation also preserves typings.

```
lemma rbeta_typing:
  ⟦ e1 →↠β e2; A ⊢ e1 : t ⟧ ⟹ A ⊢ e2 : t
```

5 Semantics

5.1 Models of Simply Typed Lambda Calculus

The theory module `FrameModel` defines what it means to be a model of simply typed lambda calculus. To accomplish this, we have chosen to use Isabelle's proof locale mechanism [12].

A proof locale is basically a way to encapsulate a set of assumptions and local definitions, so that they can be shared by many proofs. Locales can also be combined and extended with extra constants and assumptions. When you prove a theorem inside a locale, you can refer to the locale's assumptions and definitions by name during the proof. When a theorem proved inside a locale is used outside the locale, the locale assumptions turn into extra premises.

Ohori [1, Section 3.1] defines a frame as a type-indexed family of sets $\mathcal{D}$, and a family of binary operators $\bullet_{\tau_1,\tau_2} : (\mathcal{D}_{\tau_1\to\tau_2} \times \mathcal{D}_{\tau_1}) \to \mathcal{D}_{\tau_2}$. We formalize this with a proof locale as follows:

```
locale frame =
  fixes app    :: α ⇒ α ⇒ α (infixl •)
   and D       :: typ ⇒ α set
   and v_env :: typ env ⇒ (α env) set
assumes app_type:
```

```
  ⟦ f ∈ D(t1 → t2); x ∈ D(t1) ⟧ ⟹ f • x ∈ D(t2)
defines FA_env_def:
  v_env A ≡ {v. ∀k t. mapsto A k t ⟶
    (∃x. mapsto v k x ∧ x ∈ D(t))}
```

Now that we can refer to the set of values for each type, it makes sense to define the set of value environments that match a given type environment. We define the function `v_env` as a locale constant, so that it can refer to `D` in its definition.

Next we extend the frame locale to specify extensional frames. In an extensional frame, two functional values are equal if their outputs match on all properly-typed inputs.

```
locale extensional_frame = frame +
  assumes frame_ext:
    ⟦ f ∈ D(t1 → t2); g ∈ D(t1 → t2);
      ∀x ∈ D(t1). f • x = g • x ⟧ ⟹ f = g
```

Finally, we extend this locale again to define what it means to be a model of simply typed lambda calculus. A model consists of a meaning function M that satisfies several properties, listed below.

```
locale model = extensional_frame +
  fixes M :: [typ env × TL × typ, α env] ⇒ α
assumes var: ⟦ mapsto A x t; v ∈ v_env A ⟧
               ⟹ mapsto v x (M(A,Var x,t)v)
    and app: ⟦ A ⊢ e : s → t; A ⊢ f : s; v ∈ v_env A ⟧
               ⟹ M(A,App e f,t)v
                 = M(A,e,s→t)v • M(A,f,s)v
    and abs: ⟦ A{0:=s} ⊢ e : t; v ∈ v_env A; d ∈ D(s) ⟧
               ⟹ M(A,Abs s e,s→t)v • d
                 = M(A{0:=s},e,t)(v{0:=d})
    and abs_typ: ⟦ A{0:=s} ⊢ e : t; v ∈ v_env ⟧
               ⟹ M(A,Abs s e,s→t)v ∈ D(s→t)
```

With the model locale, the benefits of using Isabelle's locale mechanism become very evident. If we had decided not to use proof locales, then each of the above rules would have to be explicitly encoded in the assumptions of each theorem where they are needed. Locales greatly facilitate the management of this large set of assumptions.

After defining the model locale, we prove several theorems about models. Most of these theorems require a significant amount of manual intervention, except `rbeta_M`, which follows almost immediately from `beta_M`. The result about type correctness of the meaning function is proved by induction over the typing derivation of `A ⊢e : t`. The later theorems also proceed by induction, each making use of previous theorems.

```
lemma (in model) type_correct_M:
```

```
  ⟦ A ⊢ e : t; v ∈ v_env ⟧ ⟹ M(A,e,t)v ∈ D(t)
lemma (in model) subst_M:
  ⟦ A{k:=t'} ⊢ e : t; A ⊢ e' : t' v ∈ v_env A ⟧
    ⟹ M(A{k:=t'},e,t)(v{k:=M(A,e',t')v})
      = M(A,subst e e' k,t)v)
lemma (in model) beta_M:
  ⟦ e →β e'; A ⊢ e : t; v ∈ v_env A ⟧
   ⟹ M(A,e,t)v = M(A,e',t)v
lemma (in model) rbeta_M:
  ⟦ e →→β e'; v ∈ v_env; A ⊢ e : t ⟧
    ⟹ M(A,e,t)v = M(A,e',t)v
```

5.2 Weak Normalization

The Reduce theory module concerns the Weak Normalization property for the simply typed lambda calculus. This property means that for any typable term, there exists a reduction path that ends in a beta normal term. If the simply typed lambda calculus does indeed have the Weak Normalization property, then there must exist a function that returns an appropriate normal form for any typable input term. We state the required properties of such a function here.

```
consts reduce :: TL ⇒ TL
lemma reduce_rbeta: e →→β reduce e
lemma reduce_beta_normal:
  A ⊢ e : t ⟹ reduce e ∈ beta_normal
```

For the purposes of the rest of the framework, it would be sufficient to merely state the above lemmas as axioms. However, to prevent introducing inconsistencies, it is preferable to avoid adding axioms whenever possible. Besides, the definition of the reduce function and the correctness proof are interesting in their own right, and they make good case study of various definitional and proof facilities of Isabelle. The appendices give the full development.

5.3 Relationship between TL and Core-ML

The theory module `TL_CoreML` establishes the relationships among the simply typed lambda calculus, the untyped lambda calculus, and Core-ML. First we demonstrate syntactic relationships by defining some conversion functions: We can convert from Core-ML to untyped lambda calculus by expanding all of the let expressions, and we can convert from simply typed to untyped lambda calculus by erasing all of the type labels.

```
consts
  letexpd :: expr ⇒ dB
  erasure :: TL ⇒ dB
primrec
  letexpd (Var x) = dB.Var x
```

```
  letexpd (Abs e) = dB.Abs (letexpd e)
  letexpd (App e1 e2) = dB.App (letexpd e1) (letexpd e2)
  letexpd (LET e2 e1) = (letexpd e1)[letexpd e2/0]
primrec
  erasure (TL.Var x) = dB.Var x
  erasure (TL.Abs t e) = dB.Abs (erasure e)
  erasure (TL.App u v) = dB.App (erasure u) (erasure v)
```

After making these definitions, we prove some straightforward results about `letexpd` and `erasure`. By induction over Core-ML terms, both functions map over `subst`. In addition, we prove that erasure preserves beta reduction, using induction over the beta relation.

lemma `erasure_beta:`
`E` $\rightarrow_\beta$ `E'` $\Longrightarrow$ `erasure E` $\rightarrow_\beta$ `erasure E'`

Finally, we get to the central theorems of Ohori's paper [1, Section 3]. The following theorem expresses a correspondence between typing judgements in Core-ML and simply typed lambda calculus. The proof is automatic, by induction over typing derivations.

theorem `Theorem5:`
`(A,e,t)` $\in$ `ML.typing` $\Longrightarrow$
$\exists$`E. (A,E,t)` $\in$ `TL.typing` $\wedge$ `erasure E = letexpd e`

Next is Ohori's Theorem 6. If Theorem 5 is like an existence theorem, then Theorem 6 is basically like the corresponding uniqueness theorem: It proves that if multiple typed lambda terms satisfy Theorem 5, then they must be beta convertible. But first we need a couple of lemmas.

lemma `erasure_normal:`
`normal TLBeta.beta E` $\Longrightarrow$ `normal Lambda.beta (erasure E)`

The `erasure_normal` lemma corresponds to Ohori's Lemma 1. It expresses the connection between beta reduction of typed and untyped lambda terms. After unfolding the definition of `normal`, the proof is automatic by induction over the beta relation.

The proof of the next lemma is a bit more complicated.

lemma `erasure_reduce_eq:`
$[\![$ `A` $\vdash$ `E : t; A` $\vdash$ `E' : t; erasure (reduce E) = erasure (reduce E')` $]\!]$ $\Longrightarrow$ `reduce E = reduce E'`

The proof uses the `beta_normal_var_app.induct` rule from Section 3. We generalize the theorem to form an induction hypothesis: For typed lambda terms `E` in beta normal form, we show that for any `E'` with the same type erasure, if `A` $\vdash$`E : t` and `A` $\vdash$`E' : t` for some `A` and `t`, then `E = E'`. Furthermore, for any

E is also a variable or application, if A ⊢E : t and A ⊢E' : t' for some A, t, and t', then E = E' and t=t'. The theorem follows from the induction result by reduce_beta_normal.

In addition to these lemmas, the proof of Theorem 6 also relies on the confluence of beta reduction for the untyped lambda calculus, which is conveniently proved in the Isabelle libraries [2]. With the confluence result, we can easily prove that if an untyped term reduces to two normal forms, then they must be equal to each other.

Finally, we can now prove Theorem 6. The proof starts by applying erasure_reduce_eq, and then we use the confluence result together with erasure_rbeta, reduce_rbeta, and erasure_normal to prove that the erasures are equal.

```
theorem Theorem6:
  ⟦ (A,E,t) ∈ TL.typing; (A,E',t) ∈ TL.typing;
     erasure E = erasure E' ⟧ ⟹ reduce E = reduce E'
```

5.4 Semantics of Core-ML

The theory module MLSemantics defines the actual meaning function for the semantics of Core-ML. The meaning function takes four arguments: The first argument is the meaning function for simply typed lambda calculus. The second and third arguments are the type and value environments, and the final argument is a Core-ML term.

```
types α model_type = [typ env × TL × typ, α env] ⇒ α
consts meaning ::
  [α model_type, typ env, α env, expr] ⇒ (typ × α) set
defs
  mean-
ing M A v e ≡ { (t, M(A,E,t)v) |E t. (A,E,t) ∈ TLTyp-
ing.typing ∧ erasure E = letexpd e }
```

Finally, we can apply Theorems 5 and 6 to show that the meaning function satisfies the desired properties that we mentioned in Section 2.

```
lemma (in model) single_valued_meaning:
  v ∈ v_env A ⟹ single_valued (meaning M A v e)
lemma (in model) type_correct_meaning:
  ⟦ v ∈ v_env A; A ⊢ e : t ⟹ ∃x∈D(t). (t,x) ∈ mean-
ing M A v e
```

6 Conclusion

The end result of this formalization effort is a meaning function for Core-ML manifested as an ordinary Isabelle function. The theories developed in this

project should be able to serve as a starting point for later work in formalizing other features of Haskell's type system.

There are several shortcomings of this work. First of all, there is a lot of repetition in the theories. Many modules are near-copies of each other. It would be beneficial to make some abstractions and increase proof-reuse. Secondly, the meaning function is not immediately usable because we have no model of simply typed lambda calculus in Isabelle. We really need to be able to construct an infinite tower of function spaces as a datatype, and Isabelle/HOL does not have this capability. One possibility is to restrict all function spaces to be continuous, and then use the domain package of Isabelle/HOLCF to emulate a tower of function spaces.

There are many opportunities for immediate future work as well. This formalization only covers part of Ohori's paper. The next step would be to extend Core-ML with polymorphic constants and recursive datatypes.

6.1 Acknowledgments

I would like to thank my advisor, John Matthews, for many stimulating discussions on this subject; and most of all for getting me interested in theorem provers in the first place.

References

[1] Atsushi Ohori. A simple semantics for ML polymorphism. Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture, November 1990.

[2] Tobias Nipkow. More Church-Rosser proofs (in Isabelle/HOL). Automated Deduction: CADE-13, LNCS 1104, 1996, 733-747.

[3] Wolfgang Naraschewski and Tobias Nipkow. Type inference verified: Algorithm W in Isabelle/HOL. Types for Proofs and Programs: Intl. Workshop TYPES 1996, LNCS 1512, 1998, 317-332.

[4] Philip Wadler. Theorems for free! Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture, September 1989.

[5] Patricia Johann and Janis Voigtländer. Free theorems in the presence of seq. Proceedings of the Principles of Programming Languages, January 2004.

[6] Felix Joachimski and Ralph Matthes. Short proofs of normalization. Submitted to the Archive for Mathematical Logic, 1998.

[7] N. G. de Bruijn. Lambda calculus notation with nameless dummies. Indagationes Mathematicae, 34:381–392, 1972.

[8] Robin Milner. A theory of type polymorphism in programming. Journal of Computer and System Sciences, 17:348–375, 1978.

[9] Patricia Johann. Short-cut fusion: Proved and improved. SAIG 2001, LNCS 2196, 47–71, 2001.

[10] John Launchbury and Simon Peyton-Jones. State in Haskell. Lisp and Symbolic Computation, Special issue on state in programming languages, 293–341, 1995.

[11] Nipkow, Paulson and Wenzel. Isabelle/HOL: A proof assistant for higher order logic. Springer-Verlag, May 2003.

[12] Kammüller, Wenzel and Paulson. Locales: A sectioning concept for Isabelle.

A Proof of Weak Normalization

The strong normalization property that we want to prove is that for every typable term e, there exists a beta normal term e' such that $e \twoheadrightarrow_\beta e'$. The structure of the proof is first to define an explicit witness function that is defined on all inputs, and then to prove that on typable inputs, the output is in beta normal form.

A.1 Definition of reduction function

To define reduction, we will use a combination of two functions. The main function R maps terms to terms; the helper function T is similar, but takes an additional type argument.

The main function can be defined by primitive recursion: It only calls itself recursively with sub-terms of its argument. As long as the helper function T always terminates, R will always terminate.

$$R(x) = x \tag{1}$$

$$R(\lambda x : \tau.e) = \lambda x : \tau.R(e) \tag{2}$$

$$R(e_1\ e_2) = \begin{cases} T(\tau, e[R(e_2)/x]) & \text{if } R(e_1) = \lambda x : \tau.e, \\ R(e_1)\ R(e_2) & \text{otherwise.} \end{cases} \tag{3}$$

To define the helper function T, we will use well-founded recursion. This function takes two parameters: a type and a term. Recursive calls will either replace the type with one of its argument types, or they will keep the same type and use a smaller term. Thus the pair of arguments always decreases according

to a lexicographic ordering.

$$T(\tau, x) = x \tag{4}$$

$$T(\tau, \lambda x : \tau_x.e) = \lambda x : \tau_x.T(\tau, e) \tag{5}$$

$$T(\tau, e_1\ e_2) = \begin{cases} T(\tau_x, e[T(\tau, e_2)/x]) & \text{if } T(\tau, e_1) = \lambda x : \tau_x.e \text{ and } \tau_x \prec \tau, \\ T(\tau, e_1)\ T(\tau, e_2) & \text{otherwise.} \end{cases} \tag{6}$$

By induction on terms we can easily prove the following lemma, which states that T is the identity on beta normal terms:

$$\frac{\text{bnf}(e)}{T(\tau, e) = e} \tag{7}$$

A.2 R does beta reduction

First we will prove that T does beta reduction, using induction over the function definition.

$$e \twoheadrightarrow_\beta T(\tau, e) \tag{8}$$

Most cases are very straightforward, except the one that does substitution. We are left with the following subgoal:

$$\frac{\begin{array}{c} T(\tau, e_1) = \lambda x : \tau_x.e \\ e_1 \twoheadrightarrow_\beta T(\tau, e_1) \\ e_2 \twoheadrightarrow_\beta T(\tau, e_2) \\ e[T(\tau, e_2)/x] \twoheadrightarrow_\beta T(\tau_x, e[T(\tau, e_2)/x]) \end{array}}{e_1\ e_2 \twoheadrightarrow_\beta T(\tau_x, e[T(\tau, e_2)/x])}$$

By parallel beta reduction, we have

$$e_1\ e_2 \twoheadrightarrow_\beta T(\tau, e_1)\ T(\tau, e_2)$$

Then by substitution, this becomes

$$e_1\ e_2 \twoheadrightarrow_\beta (\lambda x : \tau.e)\ T(\tau_0, e_2)$$

By transitivity of beta, and the beta-substitution rule, we have

$$e_1\ e_2 \twoheadrightarrow_\beta e[T(\tau_0, e_2)/x]$$

Finally, by transitivity together with the last premise, the goal is proved. The proof for R is similar.

Combining these results with Lemma (8), we can conclude that R and T preserve typings:

$$\frac{\mathcal{A} \triangleright e : \tau}{\mathcal{A} \triangleright R(e) : \tau \wedge \mathcal{A} \triangleright T(\tau', e) : \tau} \tag{9}$$

A.3 Reduction lemma

Before we prove the final result about R, first we need to prove some properties about T. As a lemma, we will inductively prove the conjunction of the two following statements:

$$\frac{\mathcal{A}\{x' := \tau'\} \triangleright e : \tau,\ \mathcal{A} \triangleright e' : \tau',\ \text{bnf}(e'),\ \text{bnf}(e)}{\text{bnf}(T(\tau', e[e'/x']))} \tag{10}$$

$$\frac{\mathcal{A}\{x' := \tau'\} \triangleright e : \tau,\ \mathcal{A} \triangleright e' : \tau',\ \text{bnf}(e'), \text{va}(e)}{\text{va}(T(\tau', e[e'/x'])) \vee (\tau \sqsubseteq \tau')} \tag{11}$$

The proof is by a combination of argument type induction on τ' and structural induction on e. In proving each case, we are allowed to assume that the inductive hypotheses hold for a smaller τ' and *any* e, or with the same τ' and a smaller e. The other variables $\mathcal{A}$, x', τ, and e' are universally quantified. Note that this is essentially the same kind of induction that we used to define T.

Next we proceed by cases on e.

A.3.1 Case x

Say the term is a variable x. By Rule (**??**) we have $\text{bnf}(x)$ and $\text{va}(x)$. We need to show the following:

$$\frac{\mathcal{A}\{x' := \tau'\} \triangleright x : \tau,\ \mathcal{A} \triangleright e' : \tau',\ \text{bnf}(e'),\ \text{bnf}(x)}{\text{bnf}(T(\tau', x[e'/x']))} \tag{12}$$

$$\frac{\mathcal{A}\{x' := \tau'\} \triangleright x : \tau,\ \mathcal{A} \triangleright e' : \tau',\ \text{bnf}(e'),\ \text{va}(x)}{\text{va}(T(\tau', x[e'/x'])) \vee (\tau \sqsubseteq \tau')} \tag{13}$$

There are two cases to consider: Either $x = x'$ or $x \neq x'$. If they are not equal, then the substitution has no effect: $x[e'/x'] = x$. Thus we have $T(\tau', x[e'/x']) = T(\tau', x) = x$, which is in beta normal form. This satisfies both (12) and (13).

If the variables are equal, then $x[e'/x'] = e'$. Since e' is in beta normal form, then $T(\tau', e') = e'$, by Lemma (7). This takes care of (12). Then by the typing rules, $\mathcal{A}\{x' := \tau'\} \triangleright x : \tau$ means that $\mathcal{A}\{x' := \tau'\}(x) = \tau$, so if $x = x'$ then $\tau' = \tau$. Therefore $\tau \sqsubseteq \tau'$, which satisfies (13).

A.3.2 Case $\lambda x : \tau_1.e$

Say the term is a lambda abstraction $\lambda x : \tau_1.e$. By Rule (**??**) $\text{va}(\lambda x : \tau_1.e)$ is false, so we can immediately discharge the second subgoal. Our remaining goal is

$$\frac{\mathcal{A}\{x' := \tau'\} \triangleright (\lambda x : \tau_1.e) : \tau,\ \mathcal{A} \triangleright e' : \tau',\ \text{bnf}(e'),\ \text{bnf}(\lambda x : \tau_1.e)}{\text{bnf}(T(\tau', (\lambda x : \tau_1.e)[e'/x']))} \tag{14}$$

By the typing rule for lambda abstractions, $\mathcal{A}\{x' := \tau'\} \triangleright (\lambda x : \tau_1.e) : \tau$ means that $\mathcal{A}\{x' := \tau', x := \tau_1\} \triangleright e : \tau_2$, where $\tau = \tau_1 \to \tau_2$. We will use the following instance of our induction hypothesis (10):

$$\frac{\mathcal{A}\{x := \tau_1, x' := \tau'\} \triangleright e : \tau_2,\ \mathcal{A}\{x := \tau_1\} \triangleright e' : \tau',\ \text{bnf}(e'),\ \text{bnf}(e)}{\text{bnf}(T(\tau', e[e'/x']))}$$

All premises of the induction hypothesis are satisfied by the typing rules and simplification; thus we have $\text{bnf}(T(\tau', e[e'/x']))$. Finally, by simplification we can also show that $\text{bnf}(T(\tau', (\lambda x : \tau_1.e)[e'/x'])) = \text{bnf}(T(\tau', e[e'/x']))$.

A.3.3 Case $e_1\ e_2$

Say the term is an application $e_1\ e_2$. We need to show the following:

$$\frac{\mathcal{A}\{x' := \tau'\} \triangleright e_1\ e_2 : \tau,\ \mathcal{A} \triangleright e' : \tau',\ \text{bnf}(e'),\ \text{bnf}(e_1\ e_2)}{\text{bnf}(T(\tau', (e_1\ e_2)[e'/x']))} \tag{15}$$

$$\frac{\mathcal{A}\{x' := \tau'\} \triangleright e_1\ e_2 : \tau,\ \mathcal{A} \triangleright e' : \tau',\ \text{bnf}(e'),\ \text{va}(e_1\ e_2)}{\text{va}(T(\tau', (e_1\ e_2)[e'/x'])) \vee (\tau \sqsubseteq \tau')} \tag{16}$$

First, we can apply the typing rules: $\mathcal{A}\{x' := \tau'\} \triangleright e_1\ e_2 : \tau$ means that for some τ_1, $\mathcal{A}\{x' := \tau'\} \triangleright e_1 : \tau_1 \to \tau$ and $\mathcal{A}\{x' := \tau'\} \triangleright e_2 : \tau_1$. We will use the following instances of our induction hypotheses:

$$\frac{\mathcal{A}\{x' := \tau'\} \triangleright e_1 : \tau_1 \to \tau,\ \mathcal{A} \triangleright e' : \tau',\ \text{bnf}(e'),\ \text{bnf}(e_1)}{\text{bnf}(T(\tau', e_1[e'/x']))} \tag{17}$$

$$\frac{\mathcal{A}\{x' := \tau'\} \triangleright e_1 : \tau_1 \to \tau,\ \mathcal{A} \triangleright e' : \tau',\ \text{bnf}(e'),\ \text{va}(e_1)}{\text{va}(T(\tau', e_1[e'/x'])) \vee ((\tau_1 \to \tau) \sqsubseteq \tau')} \tag{18}$$

$$\frac{\mathcal{A}\{x' := \tau'\} \triangleright e_2 : \tau_1,\ \mathcal{A} \triangleright e' : \tau',\ \text{bnf}(e'),\ \text{bnf}(e_2)}{\text{bnf}(T(\tau', e_2[e'/x']))} \tag{19}$$

The premises of each of these inductive hypotheses are satisfied by the typing rules and simplification. Thus, we know that $T(\tau', e_1[e'/x'])$ and $T(\tau', e_2[e'/x'])$ are both in beta normal form. Furthermore, we also know that if $T(\tau', e_1[e'/x'])$ is a lambda abstraction, then $\tau_1 \to \tau$ is a range type of τ'.

If $T(\tau', e_1[e'/x'])$ is a variable or an application, then $T(\tau', (e_1\ e_2)[e'/x'])$ simplifies to $T(\tau', e_1[e'/x'])\ T(\tau', e_2[e'/x'])$, which is in beta normal form by simplification. This satisfies both (15) and (16).

If $T(\tau', e_1[e'/x'])$ is a lambda abstraction, then from (18) above we know that $(\tau_1 \to \tau)$ is a range type of τ'. Therefore, by Lemma (**??**) $\tau_1 \prec \tau'$ and $\tau \sqsubseteq \tau'$. This satisfies (16).

Next, we will show that the type label τ_x must equal τ_1. By Lemma (**??**), $\mathcal{A}\{x' := \tau'\} \triangleright e_1 : \tau_1 \to \tau$ together with $\mathcal{A} \triangleright e' : \tau'$ implies that $\mathcal{A} \triangleright e_1[e'/x'] : \tau_1 \to \tau$. Then by Lemma (9), this in turn means that $\mathcal{A} \triangleright T(\tau', e_1[e'/x']) : \tau_1 \to \tau$, that is, $\mathcal{A} \triangleright (\lambda x : \tau_x.e) : \tau_1 \to \tau$. Finally, by the typing rules, $\tau_x = \tau_1$.

In this case, $T(\tau', (e_1\ e_2)[e'/x'])$ simplifies to $T(\tau_1, e[T(\tau', e_2[e'/x'])/x])$. Then we will use an instance of the inductive hypothesis (10) that replaces τ' with its argument type τ_1:

$$\frac{\mathcal{A}\{x := \tau_1\} \triangleright e : \tau,\ \mathcal{A} \triangleright T(\tau', e_2[e'/x']) : \tau_1 \\ \text{bnf}(T(\tau', e_2[e'/x'])),\ \text{bnf}(e)}{\text{bnf}(T(\tau_1, e[T(\tau', e_2[e'/x'])/x]))} \tag{20}$$

The first premise is satisfied by $\mathcal{A} \triangleright (\lambda x : \tau_x.e) : \tau_1 \to \tau$ and the typing rules. From $\mathcal{A}\{x' := \tau'\} \triangleright e_2 : \tau_1$ and $\mathcal{A} \triangleright e' : \tau'$, Lemma (**??**) shows that $\mathcal{A} \triangleright e_2[e'/x'] : \tau_1$. Then from Lemma (9) we have $\mathcal{A} \triangleright T(\tau', e_2[e'/x']) : \tau_1$, which satisfies the second premise. We have already shown the third premise to be true, and the fourth is true by bnf($\lambda x : \tau_x.e$) and Rule (**??**). The conclusion satisfies (15).

A.4 Reduction returns beta normal terms

We want to prove that for any typable term e, the value of $R(e)$ will be in beta normal form:

$$\frac{\mathcal{A} \triangleright e : \tau}{\text{bnf}(R(e))} \tag{21}$$

We proceed by induction on the typing derivation of e. Note that if a term is typable, then all of its subterms must be typable as well.

For variables x, $R(x) = x$, which is clearly in beta normal form.

For lambda abstractions $\lambda x : \tau.e$, the inductive hypothesis tells us that $R(e)$ is in beta normal form. Then $R(\lambda x : \tau.e)) = \lambda x : \tau.R(e)$, which is also in beta normal form.

For applications $(e_1\ e_2)$, the inductive hypothesis tells us that $R(e_1)$ and $R(e_2)$ are both in beta normal form. Also, the typing rules say that if $\mathcal{A} \triangleright e_1\ e_2 : \tau$ then there exists some τ_1 such that $\mathcal{A} \triangleright e_1 : \tau_1 \to \tau$ and $\mathcal{A} \triangleright e_2 : \tau_1$. Since R preserves typings, we also know that $\mathcal{A} \triangleright R(e_1) : \tau_1 \to \tau$.

Now we have a few cases to consider: $R(e_1)$ may be is a variable, an application, or a lambda abstraction. If $R(e_1)$ is a variable or application, then $R(e_1\ e_2) = R(e_1)\ R(e_2)$, which is in beta normal form.

Otherwise, $R(e_1)$ is a lambda abstraction $\lambda x : \tau_x.e$. Since $\mathcal{A} \triangleright (\lambda x : \tau_x.e) : \tau_1 \to \tau$, the typing rules tell us that $\tau_x = \tau_1$ and $\mathcal{A}\{x := \tau_1\} \triangleright e : \tau$. Then the definition of R tells us that $R(e_1\ e_2) = T(\tau_1, e[R(e_2)/x])$. To prove that this is in beta normal form, we will use this instance of Lemma (10):

$$\frac{\mathcal{A}\{x := \tau_1\} \triangleright e : \tau,\ \mathcal{A} \triangleright R(e_2) : \tau_1,\ \text{bnf}(R(e_2)),\ \text{bnf}(e)}{\text{bnf}(T(\tau_1, e[R(e_2)/x]))}$$

We have already shown the first and third premises to be true. The second premise follows from $\mathcal{A} \triangleright e_2 : \tau_1$ and Lemma (9). The fourth premise follows from bnf($\lambda x : \tau_x.e$) and Rule (**??**). Thus the theorem is proved.

A.5 Related work

Compared with other normalization proofs, the proof presented here is most similar to one by Joachimski and Matthes [6]. The main similarity is that they use the same kind of nested induction: wellfounded induction over the size of types together with structural induction over terms. However, there are many differences that make this approach unique. First of all, this proof is unique in using wellfounded recursion to define an explicit witness function. This is in accord with a basic paradigm of Isabelle, which is that carefully designed definitions can greatly reduce the complexity of proofs [11, p. 182]. Another difference is that Joachimski and Matthes define specialized notation for doing various operations on lists of terms, such as substitution, typing judgments, application, etc. While this is relatively inexpensive in a paper proof, this adds a great deal of stuff to a formalization. The Isabelle 2003 distribution comes with a formalized version of Joachimski and Matthes' proof, ported to Isabelle by Stefan Berghofer; partially from formalizing all of the list stuff, that proof uses about 4-5 times the number of lines of code and nearly 7 times the number of tactic invocations as my proof.