A Simple Semantics for ML Polymorphism*

Atsushi Ohori

Department of Computer and Information Science,
University of Pennsylvania,
200 South 33rd Street
Philadelphia, PA 19104-6389

Abstract

We give a framework for denotational semantics for the polymorphic "core" of the programming language ML. This framework requires no more semantic material than what is needed for modeling the *simple* type discipline. In our view, terms of ML are pairs consisting of a raw (untyped) lambda term and a type-scheme that ML's type inference system can derive for the raw term. We interpret a type-scheme as a set of simple types. Then, given *any* model $\mathcal{M}$ of the simply typed lambda calculus, the meaning of an ML term will be a set of pairs, each consisting of a simple type τ and an element of $\mathcal{M}$ of type τ.

Hence, there is no need to interpret all raw terms, as was done in Milner's original semantic framework. In comparison to Mitchell and Harper's analysis, we avoid having to provide a very large type universe in which generic type-schemes are interpreted. Also, we show how to give meaning to ML terms rather than to derivations in the ML type inference system (which can be infinitely many for a single ML term).

We give an axiomatization for the equational theory that corresponds to our semantic framework and prove the analogs of the completeness theorems that Friedman proved for the simply typed lambda calculus. The framework can be extended to languages with constants, type constructors and recursive types (via regular trees). For the extended language, we prove a theorem that allows the transfer of certain full abstraction results from languages based on the typed lambda calculus to ML-like languages.

* This research was supported in part by grants NSF IRI86-10617, ARO DAA6-29-84-k-0061, and by OKI Electric Industry Co., Japan.

Appeared in **Proceedings of ACM/IFIP Conference on Functional Programming Language and Computer Architecture. Pages 281 – 292, 1989.**

1 Introduction

ML is a strongly typed programming language sharing with other typed languages the property that the type correctness of a program is completely checked by static analysis of the program – usually done at compile time. Among other strongly typed languages, one feature that distinguishes ML is its *implicit* type system. Unlike explicitly-typed languages such as Algol and Pascal, ML does not require type specifications of bound variables. The syntax of ML programs is therefore the same as that of untyped terms (raw terms). However, not all raw terms correspond to legal ML programs. A term of ML is an associations of a raw term and a type-scheme determined by ML's type inference system. Moreover, for any given raw term, the type system infers its *most general* (or *principal*) type-scheme representing the set of all possible types of the raw term. For example, the type system infers the type-scheme $t \to t$ for the raw term $\lambda x.\, x$, where t is a type variable representing arbitrary types. The above type-scheme correctly represents the set of all possible types of the raw term $\lambda x.x$. Through this type inference mechanism, ML achieves much of the flexibility and convenience of untyped languages without sacrificing the desired property of statically typed languages. In the above example, the term $\lambda x.\, x$ can safely be used as a term of any type of the form $\tau \to \tau$. By combining with the binding mechanism of ***let***-expressions, ML also realizes a form of *polymorphism* without using type abstraction or type application. In the body e of $\textbf{\textit{let}}\ id = \lambda x.\, x\ \textbf{\textit{in}}\ e$, each occurrence of id can be used as an identity function of a different type.

There are two major existing approaches to denotational semantics for ML polymorphism – the one by Milner [Mil78] (extended by MacQueen, Plotkin and Sethi [MPS86]) based on an untyped language and the other by Mitchell and Harper [MH88] based on an explicitly-typed language using Damas and Milner's type inference system [DM82]. As we shall suggest in this paper, however, neither of them properly explain the behavior of ML programs. Because of the implicit type system, ML

behaves differently from both untyped languages and explicitly-typed languages. In order to understand ML, we need to develop a framework for denotational semantics and equational theories that gives a precise account for ML's implicit type system. The goal of this paper is to propose such a framework. In the rest of this section, we review the two existing approaches and outline our approach.

1.1 Milner's original semantics

In [Mil78], Milner proposed a semantic framework for ML based on a semantics of an untyped language. He defined the following two classes of types:

$$\begin{array}{rcl} \tau & ::= & b \mid \tau \to \tau \\ \rho & ::= & t \mid b \mid \rho \to \rho \end{array}$$

where b stands for base types and t stands for type variables. Here we call them *types* (ranged over by τ) and *type-schemes* (ranged over by ρ) respectively. Type-schemes containing type variables represent all their substitution instances and correspond to polymorphic types. A type-scheme ρ_1 is *more general* than ρ_2 if ρ_2 is a substitution instance of ρ_1. He then gave the algorithm $\mathcal{W}$ that infers most general type-schemes for raw terms defined as follows:

$$\begin{array}{rcl} e & ::= & x \mid (\lambda x.e) \mid (e\ e) \mid \textbf{\textit{if}}\ e\ \textbf{\textit{then}}\ e\ \textbf{\textit{else}}\ e \mid \\ & & \textbf{\textit{fix}}\ x\ e \mid\ \textbf{\textit{let}}\ x = e\ \textbf{\textit{in}}\ e \end{array}$$

where x stands for variables.

For this language, he proposed a semantic interpretation for the typing judgement $e : \rho$ as the set-membership relation between the denotation of e and the denotation of ρ and showed that the type inference algorithm $\mathcal{W}$ is sound under this interpretation. The denotation of a raw term is defined as an element of a domain satisfying the following domain equation:

$$V = B_1 + \cdots + B_n + [V \to V] + \{\mathit{wrong}\}$$

where $B_1, \ldots, B_n$ are domains corresponding to base types and *wrong* represents run-time error. The denotation of a type is defined as a subset of V not containing *wrong*. The denotation of a type-scheme is defined as the intersection of the denotations of all its instance types. This semantics was extended to recursive types by MacQueen, Plotkin and Sethi [MPS86]. (See also [Hin83, Cop84] for related studies.)

This semantics explains the polymorphic nature of ML programs and verifies that ML typing discipline prevents all run-time type errors. However, this semantics does not fit the operational behavior of ML programs. As an example, consider the following two raw terms e_1 and e_2 with their principal type-schemes:

$$\begin{array}{rcl} e_1 & \equiv & \lambda x \lambda y.\, y\ :\ t_1 \to t_2 \to t_2 \\ e_2 & \equiv & \lambda x \lambda y.\, (\lambda z \lambda w.\, w)(xy)y\ :\ (t_3 \to t_4) \to t_3 \to t_3 \end{array}$$

where parentheses are omitted, assuming left association of applications. Under the call-by-name version of Milner's semantics, which is also the semantics defined by MacQueen, Plotkin and Sethi, the above two raw terms have the same meaning. Indeed, if we were to ignore their type-schemes and regard them as terms in the untyped lambda calculus, then they would be β-convertible to each other and would be regarded as equal terms. However, ML is apparently a typed language, and as terms of ML, these two behave quite differently. For example, under any avaluation strategy, the term $((e_1\ 1)\ 2)$ is evaluated to 2 but $((e_2\ 1)\ 2)$ is not even a legal term and ML compiler reports a type error. This is one of the most noticeable difference between meanings of terms and should be distinguished by any semantics. From this example, we can also see that the equality on ML programs is different from the equality on terms in the untyped lambda calculus.

Moreover, this semantics requires a model of the set of all untyped lambda terms, not only the ones that correspond to ML programs, i.e. the raw terms for which an ML type-scheme can be inferred.

1.2 Damas-Milner type inference system and Mitchell-Harper's analysis

Damas and Milner presented a proof system for ML typing judgements [DM82]. They redefined the set of types of ML as the following two classes:

$$\begin{array}{rcl} \rho & ::= & t \mid b \mid \rho \to \rho \\ \sigma & ::= & \rho \mid \forall t.\, \sigma \end{array}$$

ρ is Milner's type-scheme. We call σ a *generic type-scheme*. t in $\forall t.\, \sigma$ is a bound type variable. We write $\rho[t_1 := \rho_1, \ldots, t_n := \rho_n]$ for the type-scheme obtained from ρ by substituting t_i by ρ_i.

A generic type-scheme $\forall t_1 \cdots t_n.\ \rho$ is a *generic instance* of $\forall t'_1 \cdots t'_m.\rho'$ if each t_j is not free in $\forall t'_1 \cdots t'_m.\rho'$ and $\rho = \rho'[t'_1 := \rho_1, \ldots, t'_m := \rho_m]$ for some type-schemes $\rho_1, \ldots, \rho_m$. A type σ is *more general* than σ', $\sigma' < \sigma$, if σ' is a generic instance of σ.

A Damas-Milner *type assignment scheme* Γ is a function from a finite subset of variables to generic type-schemes. For a given function f, we write $f\{x_1 := v_1, \ldots, x_n := v_n\}$ for the function f' such that $dom(f') = dom(f) \cup \{x_1, \ldots, x_n\}$, $f'(y) = f(y)$ for any $y \neq x_i, 1 \leq i \leq n$ and $f'(x_i) = v_i, 1 \leq i \leq n$.

A Damas-Milner *typing scheme* is a formula of the form $\Gamma \rhd e : \sigma$ that is derivable in the following proof system:

$$(\text{VAR}) \quad \Gamma \rhd x : \sigma \quad \text{if } \Gamma(x) = \sigma$$

$$(\text{GEN}) \quad \frac{\Gamma \rhd e : \sigma}{\Gamma \rhd e : \forall t.\, \sigma} \quad \text{if } t \text{ not free in } \Gamma$$

$$(\text{INST}) \quad \frac{\Gamma \rhd e : \sigma}{\Gamma \rhd e : \sigma'} \quad \text{if } \sigma' < \sigma$$

$$(\text{ABS}) \quad \frac{\Gamma\{x := \rho_1\} \rhd e : \rho_2}{\Gamma \rhd \lambda x.\, e : \rho_1 \to \rho_2}$$

$$(\text{APP}) \quad \frac{\Gamma \rhd e_1 : \rho_1 \to \rho_2 \quad \Gamma \rhd e_2 : \rho_1}{\Gamma \rhd (e_1\ e_2) : \rho_2}$$

$$(\text{LET}) \quad \frac{\Gamma \rhd e_1 : \sigma \quad \Gamma\{x := \sigma\} \rhd e_2 : \rho}{\Gamma \rhd \ \boldsymbol{let}\ x = e_1\ \boldsymbol{in}\ e_2 : \rho}$$

We write $\boldsymbol{DM} \vdash \Gamma \rhd e : \sigma$ if $\Gamma \rhd e : \sigma$ is derivable in the proof system. In this formalism, ML terms are typing schemes. We call them Damas-Milner terms.

Based on this derivation system, Mitchell and Harper proposed another framework to explain implicit type system of ML [MH88]. In what follows, we shall only discuss their analysis of the "core" of ML. However, it should be mentioned that their approach also provides an elegant treatment of Standard ML's modules [HMM86].

They defined an explicitly-typed language, called Core-XML. The set of types of Core-XML is the same as those in Damas-Milner system. The set of *pre-terms* of Core-XML is given by the following abstract syntax:

$$M \quad ::= \quad x \mid (M\ M) \mid (\lambda x : \rho.\, M) \mid (M\ \rho) \mid (\lambda t.\, M) \mid \boldsymbol{let}\ x : \sigma = M\ \boldsymbol{in}\ M$$

where $(M\ \rho)$ is a type application and $(\lambda t.\, M)$ is a type abstraction. Type-checking rules for Core-XML are given as follows:

$$(\text{VAR}) \quad \Gamma \rhd x : \sigma \quad \text{if } \Gamma(x) = \sigma$$

$$(\text{TABS}) \quad \frac{\Gamma \rhd M : \sigma}{\Gamma \rhd (\lambda t.\, M) : \forall t.\, \sigma} \quad \text{if } t \text{ not free in } \Gamma$$

$$(\text{TAPP}) \quad \frac{\Gamma \rhd M : \forall t.\, \sigma}{\Gamma \rhd (M\ \rho) : \sigma[t := \rho]}$$

$$(\text{ABS}) \quad \frac{\Gamma\{x := \rho_1\} \rhd M : \rho_2}{\Gamma \rhd (\lambda x : \rho_1.\, M) : \rho_1 \to \rho_2}$$

$$(\text{APP}) \quad \frac{\Gamma \rhd M_1 : \rho_1 \to \rho_2 \quad \Gamma \rhd M_2 : \rho_1}{\Gamma \rhd (M_1\ M_2) : \rho_2}$$

$$(\text{LET}) \quad \frac{\Gamma \rhd M_1 : \sigma \quad \Gamma\{x := \sigma\} \rhd M_2 : \rho}{\Gamma \rhd \ \boldsymbol{let}\ x : \sigma = M_1\ \boldsymbol{in}\ M_2 : \rho}$$

We write $\boldsymbol{MH} \vdash \Gamma \rhd M : \sigma$ if $\Gamma \rhd M : \sigma$ is derivable from the above typing rules. Terms of Core-XML are typing schemes that are derivable in the above system. They showed the following relationship between Core-XML and Damas-Milner system. Define the *type erasure* of a pre-term M, $\boldsymbol{er}(M)$, as follows:

$$\begin{aligned}
\boldsymbol{er}(x) &= x \\
\boldsymbol{er}((M_1\ M_2)) &= (\boldsymbol{er}(M_1)\ \boldsymbol{er}(M_2)) \\
\boldsymbol{er}((\lambda x : \rho.\, M)) &= (\lambda x.\, \boldsymbol{er}(M)) \\
\boldsymbol{er}((\lambda t.\, M)) &= \boldsymbol{er}(M) \\
\boldsymbol{er}((M\ \rho)) &= \boldsymbol{er}(M) \\
\boldsymbol{er}(\ \boldsymbol{let}\ x : \sigma = M_1\ \boldsymbol{in}\ M_2) &= \\
&\quad \boldsymbol{let}\ x = \boldsymbol{er}(M_1)\ \boldsymbol{in}\ \boldsymbol{er}(M_2)
\end{aligned}$$

Theorem 1 (Mitchell-Harper) *If $\boldsymbol{MH} \vdash \Gamma \rhd M : \sigma$ then $\boldsymbol{DM} \vdash \Gamma \rhd \boldsymbol{er}(M) : \sigma$. If $\boldsymbol{DM} \vdash \Gamma \rhd e : \sigma$ then there exists a Core-XML pre-term M such that $\boldsymbol{er}(M) \equiv e$ and $\boldsymbol{MH} \vdash \Gamma \rhd M : \sigma$. Moreover, M can be computed effectively from a derivation of $\Gamma \rhd e : \sigma$.* ∎

Based on this relationship, they concluded that Core-XML and Damas-Milner system are "equivalent" and regarded ML as a "convenient shorthand" for Core-XML.

If we could indeed regard ML terms as syntactic shorthands for Core-XML terms then equational theory and model theory could be those of Core-XML. However, the above result does not establish any syntactic mapping from Damas-Milner terms to Core-XML terms. It only established a correspondence between *derivations* of Damas-Milner terms and Core-XML terms. But a Damas-Milner term has in general infinitely many distinct derivations. This means that there are, in general, infinitely many distinct Core-XML terms that correspond to a given Damas-Milner term. For example, consider the term:

$$\emptyset \rhd (\lambda x \lambda y.\, y)(\lambda x.\, x) : t \to t.$$

Any Core-XML term of the form

$$(\lambda x : \rho \to \rho \lambda y : t.\, y)(\lambda x : \rho.\, x)$$

for any type-scheme ρ corresponds to the above typing. One way to overcome this difficulty is to *choose* a particular Core-XML term among possibly infinitely many choices. Such a choice seems possible if we assume a particular type inference algorithm. But we would like to avoid such an assumption in a formal analysis of ML. Another possibility would be to consider a Damas-Milner term as an equivalence class of Core-XML terms. One plausible equivalence relation is the convertibility (or equality) relation. If a Damas-Milner term corresponds to a convertibility class of Core-XML terms then any model of Core-XML in which the convertibility relation is sound yields a semantics of Damas-Milner terms. Unfortunately, however, Damas-Milner terms do not correspond to convertibility classes of Core-XML terms. To see this, suppose we have two base types b_1, b_2 and consider the following Damas-Milner term:

$$\{x : \forall t.\, t \to b_1, y : \forall t.\, t \to t\} \rhd (x\ y) : b_1.$$

The following two Core-XML terms both correspond to derivations of the above term:

$$\{x : \forall t.\, t \to b_1, y : \forall t.\, t \to t\} \rhd ((x\ b_2 \to b_2)\ (y\ b_2)) : b_1$$

$$\{x : \forall t.\, t \to b_1, y : \forall t.\, t \to t\} \rhd ((x\ b_1 \to b_1)\ (y\ b_1)) : b_1$$

But these two terms are both in normal form and therefore are not convertible.

We also think that Damas-Milner system and the corresponding explicitly-typed language Core-XML are too strong to explain ML's type system. As argued by Milner in [Mil78], it is ML's unique feature and advantage that ML supports polymorphism without type abstraction and type application. Note that this account of ML only used non-generic type-schemes. As such a language, ML can be better understood without using generic type-schemes, whose semantics requires the construction of very large spaces.

1.3 A simple framework for ML polymorphism

From the above analyses, it appears that ML is different from both untyped languages and explicitly typed languages. In order to understand ML properly we will develop a framework for semantics that accounts for ML's implicit type system. Such a semantics should be useful to reason about various properties of ML programs including equality on programs and operational semantics. A strategy was already suggested in Mitchell-Harper approach. We can use an explicitly typed language as an intermediate language to define a semantics of ML. In this paper, we use the simply typed lambda calculus. Usage of the simply typed lambda calculus to explain ML polymorphism was suggested in Wand's analysis [Wan84], where ML terms are regarded as shorthands for terms in the simply typed lambda calculus. Wand's approach, however, shares the same difficulty as in Mitchell-Harper's analysis. It only gives meanings to derivations. Moreover, it does not deal with derivations whose type-schemes contain type variables.

We first define an inference system and semantics of ML *typings* (typing schemes that do not contain type variables) and then generalize them to ML terms (i.e. typing schemes). Analogous to the relationship between Damas-Milner system and Core-XML, derivations of typings in our system correspond to terms of the simply typed lambda calculus. Here is the crucial point in the development of our semantic approach: we show that if two typed terms correspond to derivations of a same ML typing then they are β-convertible (theorem 6). This guarantees that any model of the simply typed lambda calculus, in which the rule (β) is sound, indeed yields a semantics of ML typings. We then regard a general ML term as a representation of a *set* of typings. The denotation of an ML term is defined as the set of denotations of the typings indexed by types represented by its type-scheme. For example, we regard the denotation $[\![\emptyset \rhd \lambda x.\, x\ :\ t \to t]\!]$ as the set $\{(\tau \to \tau, [\![\lambda x : \tau.\, x]\!]) | \tau \in Type\}$.

Equational theories are defined not on raw terms but on typing schemes. Two typing schemes are equal iff their type-schemes are equal and raw terms are convertible to each other. We then prove the soundness and completeness of equational theories. This confirms that our notion of semantics precisely captures and justifies the informal intuition behind the behavior of ML programs.

Our semantic framework can be extended to languages with constants, type constructors and recursive types (via infinite regular trees). Our semantic framework can also be related to certain operational semantics. We show that if a semantics of the typed lambda calculus is fully abstract with respect to an operational semantics then the corresponding semantics of ML is fully abstract with respect to an operational semantics that satisfies certain reasonable properties in connection with the operational semantics of the typed lambda calculus. This results enables us to transfer various existing results for full abstraction of typed languages to ML-like languages. A limitation to this program is due to the fact that our interpretation needs the soundness of the (β) rule. Such models, of course, while good for "call-by-name" evaluation, are not computationally adequate for the usual "call-by-value" evaluation of ML programs. Thus, our full abstraction result seems helpful only for "lazy" ML-like languages such as Miranda [Tur85] and Lazy ML [Aug84].

2 The Language Core-ML

We first present our framework for the set of pure raw terms, the same set analyzed in [DM82, MH88]. We call the pure language Core-ML. Later in section 5 we extend our frameworks to a language allowing constants, type constructors and recursive types (infinite types).

2.1 Raw terms, types and type-schemes

We assume that we are given a countably infinite set of variables (ranged over by x). The set of *raw terms* of Core-ML (ranged over by e) is defined by the following abstract syntax:

$$e ::= x \mid (e\ e) \mid \lambda x.\, e \mid \ \textbf{\textit{let}}\ x = e\ \textbf{\textit{in}}\ e$$

We write $e[e_1/x_1, \ldots, e_n/x_n]$ for the raw term obtained from e by simultaneously replacing $x_1, \ldots, x_n$ by $e_1, \ldots, e_n$ with necessary bound variable renaming. The intended meaning of $\textbf{\textit{let}}\ x = e_1\ \textbf{\textit{in}}\ e_2$ is to *bind*

x to e_1 in e_2 and to denote operationally the expression $e_2[e_1/x]$. For a raw term e, the *let expansion* e, $letexpd(e)$, is the raw term without ***let***-expression obtained from e by repeatedly replacing the outmost subterm of the form $\boldsymbol{let}\ x = e_1\ \boldsymbol{in}\ e_2$ by $e_2[e_1/x]$. For any raw term e, $letexpd(e)$ always exists.

The set of types and type-schemes are the same as those defined in Milner's analysis.

A *substitution* θ is a function from type variables to type-schemes such that $\theta(t) \neq t$ for only finitely many t. We identify θ with its extension to type-schemes (and any other syntactic structures that contain type-schemes). A type-scheme ρ is an *instance* of a type-scheme ρ' if there is a substitution θ such that $\theta(\rho') = \rho$. If ρ is a type then it is a *ground* instance.

2.2 Typings, typing schemes and terms of Core-ML

A *type assignment* $\mathcal{A}$ is a function from a finite subset of variables to types. A *typing* is a formula of the form $\mathcal{A} \rhd e : \tau$ that is derivable in the following proof system:

$$(\text{VAR}) \quad \mathcal{A} \rhd x : \tau \quad \text{if } \mathcal{A}(x) = \tau$$

$$(\text{APP}) \quad \frac{\mathcal{A} \rhd e_1 : \tau_1 \to \tau_2 \quad \mathcal{A} \rhd e_2 : \tau_1}{\mathcal{A} \rhd (e_1\ e_2) : \tau_2}$$

$$(\text{ABS}) \quad \frac{\mathcal{A}\{x := \tau_1\} \rhd e : \tau_2}{\mathcal{A} \rhd \lambda x.\, e : \tau_1 \to \tau_2}$$

$$(\text{LET}) \quad \frac{\mathcal{A} \rhd e_1[e_2/x] : \tau \quad \mathcal{A} \rhd e_2 : \tau'}{\mathcal{A} \rhd \ \boldsymbol{let}\ x = e_2\ \boldsymbol{in}\ e_1 : \tau}$$

In the rule (LET), τ' may be any type. We write $\boldsymbol{ML} \vdash \mathcal{A} \rhd e : \tau$ if $\mathcal{A} \rhd e : \tau$ is derivable in the above proof system. A *derivation* Δ of $\mathcal{A} \rhd e : \tau$ is a proof tree for $\mathcal{A} \rhd e : \tau$ in the above proof system.

A *type assignment scheme* Σ is a function from a finite subset of variables to type-schemes. A *typing scheme* is a formula of the form $\Sigma \rhd e : \rho$ whose ground instances are all typings, i.e. $\Sigma \rhd e : \rho$ is a typing scheme if for any ground instance $(\mathcal{A}, \tau)$ of (Σ, ρ), $\boldsymbol{ML} \vdash \mathcal{A} \rhd e : \tau$. We write $\boldsymbol{ML} \vdash \Sigma \rhd e : \rho$ if $\Sigma \rhd e : \rho$ is a typing scheme. A typing scheme $\Sigma_1 \rhd e : \rho_1$ is *more general than* a typing scheme $\Sigma_2 \rhd e : \rho_2$, write $\Sigma_2 \rhd e : \rho_2 < \Sigma_1 \rhd e : \rho_1$, if there is a substitution θ such that $\Sigma_2{\uparrow}^{dom(\Sigma_1)} = \theta(\Sigma_1)$ and $\rho_2 = \theta(\rho_1)$, where $f{\uparrow}^X$ denotes the function restriction of f on X. Note that more general also means less entries in a type assignment scheme. A typing scheme $\Sigma \rhd e : \rho$ is *principal* if $\Sigma' \rhd e : \rho' < \Sigma \rhd e : \rho$ for any typing scheme $\Sigma' \rhd e : \rho'$. The following property is an immediate consequence of the definition:

Proposition 1 *If $\Sigma \rhd e : \rho$ is a principal typing scheme then $\{\mathcal{A} \rhd e : \tau | \mathcal{A} \rhd e : \tau < \Sigma \rhd e : \rho\} = \{\mathcal{A} \rhd e : \tau | \boldsymbol{ML} \vdash \mathcal{A} \rhd e : \tau\}$.* ∎

This means that a principal typing scheme represents the set of all provable typings. In what follows, we regard typing schemes as representatives of equivalence classes under the preorder $<$, which correspond to equivalence classes induced by renaming of type variables (without collapsing distinct variables).

We now define terms of ML as (not necessarily principal) typing schemes. Non principal typing schemes correspond to programs with (partial) type specifications, which are supported in ML and can be easily added to our definition. A term containing type variables corresponds to a polymorphic program in ML.

Under the above characterization, the problem of type-checking is stated as follows:

> Given a type assignment scheme Σ, a raw term e and a type-scheme ρ, determine whether $\boldsymbol{ML} \vdash \Sigma \rhd e : \rho$ or not.

The type inference problem is stated as follows:

> Given a raw term e, determine the set $\{(\Sigma, \rho) | \boldsymbol{ML} \vdash \Sigma \rhd e : \rho\}$.

The following theorem, which is essentially due to Hindley [Hin69], solves both of the problems:

Theorem 2 *If a raw term e has a typing scheme then it has a principal typing scheme. Moreover, there is an algorithm which, given a raw term, computes a principal typing scheme if one exists otherwise reports failure.*

Proof This is a simple extension of Hindley's result of principal typing schemes for untyped lambda term by using the following property: $\boldsymbol{ML} \vdash \mathcal{A} \rhd \boldsymbol{let}\ x = e_1\ \boldsymbol{in}\ e_2 : \tau$ iff $\boldsymbol{ML} \vdash \mathcal{A} \rhd e_2[e_1/x] : \tau$ and $\boldsymbol{ML} \vdash \mathcal{A} \rhd e_1 : \tau'$ for some τ'. ∎

The decidability of the type-checking problem follows from the decidability of the relation $\Sigma_1 \rhd e : \rho_1 < \Sigma_2 \rhd e : \rho_2$. The set $\{(\Sigma, \rho) | \boldsymbol{ML} \vdash \Sigma \rhd e : \rho\}$ is determined by the principal typing scheme using proposition 1.

This typing derivation system is significantly simpler than that of Damas-Milner system. In particular, it does not involve generic type-schemes. However, for closed terms, they are essentially equivalent in the sense of the following theorem. We omit its lengthy proof, which can be found in [Oho89].

Theorem 3 *For a closed raw term e, if $\boldsymbol{DM} \vdash \emptyset \rhd e : \sigma$ then $\boldsymbol{ML} \vdash \emptyset \rhd e : \rho_\sigma$, where ρ_σ is the type-scheme obtained from σ by substituting all bound type variables with fresh type variables. Conversely, if $\boldsymbol{ML} \vdash \emptyset \rhd e : \rho$ then $\boldsymbol{DM} \vdash \emptyset \rhd e : \rho$* ∎

As we have demonstrated by theorem 2 and 3, ML's syntactic properties are understood without using generic type-schemes. This correspond to our semantics which only requires the semantic space of the

simply typed lambda calculus. However, our system suggests a potentially inefficient type inference algorithm. A straightforward implementation of an algorithm based on our derivation system would infer a typing scheme of $\boldsymbol{let}\ x = e_1\ \boldsymbol{in}\ e_2$ by inferring a typing scheme of $e_2[e_1/x]$. This involves repeated inferences of a typing scheme of e_1 because of multiple occurrences of x in e_2. This repetition is clearly redundant. The extra typing rules for generic type-schemes in Damas-Milner system and the corresponding control structures of the algorithm $\mathcal{W}$ can be regarded as a mechanism to eliminate this redundancy and could be considered as implementation aspects of ML type inference.

2.3 Equational theories of Core-ML

A formula of an equational theory is a pair of terms having the same type assignment scheme and the same type-scheme. We write $\Sigma \rhd e_1 = e_2 : \rho$ for such a pair. An *ML-theory* consists of a given set of equations E_{ML} and the following set of rules: the axiom schemes $(\alpha), (\beta), (\eta)$, the inference rule scheme (ξ) obtained from respective rule schemes in the untyped lambda calculus by *tagging* Σ and ρ (and restricting axioms to the set of pairs of legal ML terms), the set of rule schemes for usual equational reasoning (i.e. reflexivity, symmetry, transitivity and congruence), the following axiom scheme:

$$(let) \qquad \Sigma \rhd (\ \boldsymbol{let}\ x = e_1\ \boldsymbol{in}\ e_2) = (e_2[e_1/x]) : \rho,$$

and the following inference rule scheme:

$$(\Sigma) \qquad \frac{\Sigma \rhd e_1 = e_2 : \rho}{\Sigma' \rhd e_1 = e_2 : \rho} \qquad \text{if } \Sigma \subseteq \Sigma' \text{ (as graphs).}$$

We require a set of equation E_{ML} to have the following property:

> $\Sigma \rhd e_1 = e_2 : \rho \in E_{\text{ML}}$ iff for any ground instance $(\mathcal{A}, \tau)$, $\mathcal{A} \rhd e_1 = e_2 : \tau \in E_{\text{ML}}$,

Intuitively, this restriction states that a set of equation should be specified by a set of most general equations. We believe that all useful equations for ML satisfy this restriction. We call a set of equation E_{ML} satisfying the above properties as a set of *ML-equations*.

We write $E_{\text{ML}} \vdash_{\text{ML}} \Sigma \rhd e_1 = e_2 : \rho$ if $\Sigma \rhd e_1 = e_2 : \rho$ is derivable from the axioms and E_{ML} using the inference rules. A set of ML-equations E_{ML} determines the ML-theory $Th_{\text{ML}}(E_{\text{ML}})$. We sometimes regard $Th_{\text{ML}}(E_{\text{ML}})$ as the set of all equations that are provable by the theory. The theory $Th_{\text{ML}}(\emptyset)$ corresponds to the equality on ML terms. We write $\Sigma \rhd e_1 =_{\text{ML}} e_2 : \rho$ for $\emptyset \vdash_{\text{ML}} \Sigma \rhd e_1 = e_2 : \rho$.

If we exclude the rule of symmetry from the set of rules, then we have the notion of *reductions*. We write $E_{\text{ML}} \vdash_{\text{ML}} \Sigma \rhd e_1 \twoheadrightarrow e_2 : \rho$ if $\Sigma \rhd e_1 : \rho$ is reducible to $\Sigma \rhd e_2 : \rho$ using E_{ML} and the set of rules. In particular, the empty set determines the $\beta\eta$-reducibility, for which we write $\mathcal{A} \rhd M_1 \twoheadrightarrow_{\text{ML}} M_2 : \tau$.

3 Semantics of Core-ML

In this section, we first define the explicitly-typed language $T\Lambda$ that corresponds to derivations of Core-ML typings. We then define the semantics of Core-ML relative to a model of $T\Lambda$.

3.1 Explicitly-typed language $T\Lambda$ and its semantics

The set of types of $T\Lambda$ is exactly those in Core-ML. The set of pre-terms is given by the following abstract syntax:

$$M ::= x \mid (M\ M) \mid \lambda x : \tau.\ M$$

Type-checking rules for $T\Lambda$ are given as follows:

$$(\text{VAR}) \quad \mathcal{A} \rhd x : \tau \quad \text{if } \Gamma(x) = \tau$$

$$(\text{ABS}) \quad \frac{\mathcal{A}\{x := \tau_1\} \rhd M : \tau_2}{\mathcal{A} \rhd (\lambda x : \tau_1.\ M) : \tau_1 \to \tau_2}$$

$$(\text{APP}) \quad \frac{\mathcal{A} \rhd M_1 : \tau_1 \to \tau_2 \qquad \mathcal{A} \rhd M_2 : \tau_1}{\mathcal{A} \rhd (M_1\ M_2) : \tau_2}$$

We write $\boldsymbol{T\Lambda} \vdash \mathcal{A} \rhd M : \tau$ if $\mathcal{A} \rhd M : \tau$ is derivable from the above typing rules. The set of terms of $T\Lambda$ is the set of all derivable typings. $T\Lambda$ is clearly a representation of the simply typed lambda calculus, whose equational theory and model theory are well understood.

We write $\mathcal{A} \rhd M_1 = M_2 : \tau$ for an equation of $T\Lambda$. A $T\Lambda$-*theory* consists of a set $E_{T\Lambda}$ of equations and the following set of rules: the axiom schemes $(\alpha), (\beta), (\eta)$ and the inference rule scheme (ξ) of the simply typed lambda calculus, the set of rule schemes for usual equational reasoning and the following inference rule scheme:

$$(\mathcal{A}) \qquad \frac{\mathcal{A} \rhd M_1 = M_2 : \tau}{\mathcal{A}' \rhd M_1 = M_2 : \tau} \quad \text{if } \mathcal{A} \subseteq \mathcal{A}' \text{ (as graphs)}$$

We write $E_{T\Lambda} \vdash_{T\Lambda} \mathcal{A} \rhd M_1 = M_2 : \tau$ if $\mathcal{A} \rhd M_1 = M_2 : \tau$ is derivable from the axioms and $E_{T\Lambda}$ using the inference rules. The following notations and notions are defined parallel to those in Core-ML: $Th_{T\Lambda}(E_{T\Lambda})$, $\mathcal{A} \rhd M_1 =_{T\Lambda} M_2 : \tau$, the notion of reductions, $E_{T\Lambda} \vdash_{T\Lambda} \mathcal{A} \rhd M_1 \twoheadrightarrow M_2 : \tau$, and $\mathcal{A} \rhd M_1 \twoheadrightarrow_{T\Lambda} M_2 : \tau$.

Following [Fri73] we define *models* of $T\Lambda$ as follows: A *frame* is a pair $(\mathcal{D}, \bullet)$ where $\mathcal{D}$ is a set $\{D_\tau | \tau \in Types\}$ such that each D_τ is non-empty and $\bullet$ is a family of binary operations $\bullet_{\tau_1,\tau_2} : D_{\tau_1 \to \tau_2} \times D_{\tau_1} \to D_{\tau_2}$. A frame is *extensional* if

$$\forall \tau_1, \tau_2 \in Types, \forall f, g \in D_{\tau_1 \to \tau_2}.$$
$$(\forall d \in D_{\tau_1}. f \bullet d = g \bullet d) \Longrightarrow f = g$$

In a frame, a map $\phi : D_{\tau_1} \to D_{\tau_2}$ is *representable* if there is some $f \in D_{\tau_1 \to \tau_2}$ such that $\forall d \in D_{\tau_1}.\phi(d) = f \bullet d$ (f is a *representative* of ϕ). In an extensional frame, representatives are unique. For a frame $\mathcal{F} = (\mathcal{D}, \bullet)$ and a type assignment $\mathcal{A}$, an $\mathcal{FA}$-environment ε is a mapping from $dom(\mathcal{A})$ to $\bigcup \mathcal{D}$ such that $\varepsilon(x) \in D_{\mathcal{A}(x)}$. We write $Env^{\mathcal{F}}(\mathcal{A})$ for the set of all $\mathcal{FA}$-environments. A semantics of a term $\mathcal{A} \rhd M : \rho$ is a mapping from $Env^{\mathcal{F}}(\mathcal{A})$ to D_ρ. An extensional frame $\mathcal{M}$ is a *model* if there is a semantic mapping $[\![\,]\!]$ on terms of $T\Lambda$ satisfying the following equations: for any $\varepsilon \in Env^{\mathcal{M}}(\mathcal{A})$,

$$[\![\mathcal{A} \rhd x : \tau]\!]\varepsilon = \varepsilon(x),$$

$$[\![\mathcal{A} \rhd \lambda x : \tau_1.\, M : \tau_1 \to \tau_2]\!]\varepsilon =$$
the representative of the function ϕ s.t. $\forall d \in D_{\tau_1}$
$\phi(d) = [\![\mathcal{A}\{x := \tau_1\} \rhd M : \tau_2]\!]\varepsilon\{x := d\}$,

$$[\![\mathcal{A} \rhd (M\ N) : \tau]\!]\varepsilon =$$
$[\![\mathcal{A} \rhd M : \tau_1 \to \tau]\!]\varepsilon \bullet [\![\mathcal{A} \rhd N : \tau_1]\!]\varepsilon$

Note that for a given extensional frame, such a semantic mapping does not always exist, but if one exists then it is unique. If $\mathcal{M}$ is a model, then we write $\mathcal{M}[\![\,]\!]$ for the unique semantic mapping.

An equation $\mathcal{A} \rhd M = N : \tau$ is *valid* in a model $\mathcal{M}$, write $\mathcal{M} \models_{T\Lambda} \mathcal{A} \rhd M = N : \tau$, if $\mathcal{M}[\![\mathcal{A} \rhd M : \tau]\!] = \mathcal{M}[\![\mathcal{A} \rhd M : \tau]\!]$. Let $\boldsymbol{Valid}^{T\Lambda}(\mathcal{M})$ be the set of all $T\Lambda$ equations that are valid in $\mathcal{M}$. Write $\mathcal{M} \models_{T\Lambda} F$ for $F \subseteq \boldsymbol{Valid}^{T\Lambda}(\mathcal{M})$. For $T\Lambda$ we have the following soundness and completeness of equational theories [Fri73]:

Theorem 4 (Friedman) *For any model $\mathcal{M}$ and any $T\Lambda$-theory $Th_{T\Lambda}(E_{T\Lambda})$, if $\mathcal{M} \models_{T\Lambda} E_{T\Lambda}$ then $Th_{T\Lambda}(E_{T\Lambda}) \subseteq \boldsymbol{Valid}^{T\Lambda}(\mathcal{M})$. For any $T\Lambda$-theory T, there exists a model $\mathcal{T}$ such that $\boldsymbol{Valid}^{T\Lambda}(\mathcal{T}) = T$.* ∎

3.2 Relationship between $T\Lambda$ and Core-ML

Analogous to the relationship between Damas-Milner system and Core-XML, derivations of Core-ML typings correspond to terms of $T\Lambda$. Define a mapping $\boldsymbol{t\lambda}$ on derivations of Core-ML typings as follows:

(1) If Δ is the one node derivation tree

$$\mathcal{A} \rhd x : \tau \qquad \text{(VAR)}$$

then $\boldsymbol{t\lambda}(\Delta) = x$.

(2) If Δ is the tree of the form

$$\frac{\Delta_1}{\mathcal{A} \rhd \lambda x.\, e_1 : \tau_1 \to \tau_2} \qquad \text{(ABS)}$$

then $\boldsymbol{t\lambda}(\Delta) = \lambda x : \tau_1.\, \boldsymbol{t\lambda}(\Delta_1)$.

(3) If Δ is the tree of the form

$$\frac{\Delta_1 \qquad \Delta_2}{\mathcal{A} \rhd (e_1\ e_2) : \tau} \qquad \text{(APP)}$$

then $\boldsymbol{t\lambda}(\Delta) = (\boldsymbol{t\lambda}(\Delta_1)\ \boldsymbol{t\lambda}(\Delta_2))$.

(5) If Δ is the tree of the form

$$\frac{\Delta_1 \qquad \Delta_2}{\mathcal{A} \rhd \boldsymbol{let}\ x = e_1\ \boldsymbol{in}\ e_2 : \tau} \qquad \text{(LET)}$$

then $\boldsymbol{t\lambda}(\Delta) = \boldsymbol{t\lambda}(\Delta_1)$.

The *type erasure* of a pre-term M, write $\boldsymbol{er}(M)$, is the raw term obtained from M by *erasing* all type specifications of the form "$: \tau$" in all subterms of the form $\lambda x : \tau.\, M'$ in M. The following theorem corresponds to theorem 1:

Theorem 5 *If $\boldsymbol{T\Lambda} \vdash \mathcal{A} \rhd M : \tau$ then there is a derivation Δ for $\mathcal{A} \rhd \boldsymbol{er}(M) : \tau$ such that $\boldsymbol{t\lambda}(\Delta) \equiv M$. If Δ is a derivation of $\mathcal{A} \rhd e : \tau$ then $letexpd(e) \equiv \boldsymbol{er}(\boldsymbol{t\lambda}(\Delta))$ and $\boldsymbol{T\Lambda} \vdash \mathcal{A} \rhd \boldsymbol{t\lambda}(\Delta) : \tau$.*

Proof These properties are shown by inductions on the structures of M and the height of Δ respectively. ∎

Different from Core-XML and Damas-Milner system, we also have the following desired property:

Theorem 6 *If Δ_1, Δ_2 are derivations of a same typing $\mathcal{A} \rhd e : \tau$ then the following equation holds:*

$$\mathcal{A} \rhd \boldsymbol{t\lambda}(\Delta_1) =_{T\Lambda} \boldsymbol{t\lambda}(\Delta_2) : \tau.$$

Proof The proof uses the following lemmas:

Lemma 1 *Let $\mathcal{A} \rhd M : \tau$ and $\mathcal{A} \rhd e : \tau$ be respectively $T\Lambda$ term and Core-ML term such that $\boldsymbol{er}(M) \equiv e$. If $\mathcal{A} \rhd M \twoheadrightarrow_{T\Lambda} M' : \tau$ then there is some e' such that $\boldsymbol{er}(M') \equiv e'$ and $\mathcal{A} \rhd e \twoheadrightarrow_{\text{ML}} e' : \tau$. Conversely, if $\mathcal{A} \rhd e \twoheadrightarrow_{\text{ML}} e' : \tau$ then there is some M' such that $\boldsymbol{er}(M') \equiv e'$ and $\mathcal{A} \rhd M \twoheadrightarrow_{T\Lambda} M' : \tau$.*

Proof This is proved by observing the following facts: (1) there is a one-one correspondence between the set of $\beta\eta$-redexes in M and the set of $\beta\eta$-redexes in e, (2) if $\boldsymbol{er}((\lambda x : \tau.\, M_1)\ M_2) \equiv ((\lambda x.e_1)\ e_2)$ then $\boldsymbol{er}(M_1[M_2/x]) \equiv e_1[e_2/x]$, and (3) if $\boldsymbol{er}(\lambda x : \tau.\, Mx) \equiv (\lambda x.ex)$ then $\boldsymbol{er}(M) \equiv e$. ∎

Note that this result, combined with the property of the reduction rule (*let*) and the connection between $T\Lambda$ terms and typing derivations of ML implies that if $T\Lambda$ has the strong normalization property then so does Core-ML, which was suggested in [HS86, remark 15.32]. Technical difficulty of treating bound variables mentioned in [HS86, remark 15.32] was overcome by our presentation of $T\Lambda$.

Lemma 2 *If two terms $\mathcal{A} \rhd M_1 : \tau$ and $\mathcal{A} \rhd M_2 : \tau$ are in β-normal form and $\boldsymbol{er}(M_1) \equiv \boldsymbol{er}(M_2)$ then $M_1 \equiv M_2$.*

Proof The proof is by induction on the structure of M_1. Basis is trivial. Induction step is by cases.

Case of $M_1 \equiv \lambda x : \tau_1. M_1'$: By the typing rules, $\boldsymbol{T\Lambda} \vdash \mathcal{A}\{x := \tau_1\} \rhd M_1' : \tau_2$ for some τ_2 and $\tau = \tau_1 \to \tau_2$. Since $\boldsymbol{er}(M_1) \equiv \boldsymbol{er}(M_2)$, M_2 must be of the form $\lambda x : \tau_1'. M_2'$ such that $\boldsymbol{er}(M_1') \equiv \boldsymbol{er}(M_2')$. By the typing rules, $\boldsymbol{T\Lambda} \vdash \mathcal{A}\{x := \tau_1'\} \rhd M_2 : \tau_2'$ for some τ_2' and $\tau = \tau_1' \to \tau_2'$. Therefore $\tau_1 = \tau_1', \tau_2 = \tau_2'$. By definition, $\mathcal{A}\{x := \tau_1\} \rhd M_1' : \tau_2$ and $\mathcal{A}\{x := \tau_1\} \rhd M_2' : \tau_2$ must be also in β-normal form. Then by induction hypothesis, $M_1' \equiv M_2'$. This implies $M_1 \equiv M_2$.

Case of $M_1 \equiv (\cdots (x\ M_1^1) \cdots M_1^n)$: By the typing rules, $\boldsymbol{T\Lambda} \vdash \mathcal{A} \rhd M_1^i : \tau_1^i$ for some τ_1^i, $1 \le i \le n$. It is shown by simple induction that $\mathcal{A}(x) = \tau_1^1 \to \tau_1^2 \to \cdots \to \tau_1^n \to \tau$. Since $\boldsymbol{er}(M_1) \equiv \boldsymbol{er}(M_2)$, M_2 must be of the form $(\cdots (x\ M_2^1) \cdots M_2^n)$ and $\boldsymbol{er}(M_1^i) \equiv \boldsymbol{er}(M_2^i), 1 \le i \le n$. Then similarly we have $\boldsymbol{T\Lambda} \vdash \mathcal{A} \rhd M_2^i : \tau_2^i$ for some τ_2^i, $1 \le i \le n$ and $\mathcal{A}(x) = \tau_2^1 \to \tau_2^2 \to \cdots \to \tau_2^n \to \tau$. This implies $\tau_1^i = \tau_2^i, 1 \le i \le n$. Then by induction hypothesis, $M_1^i \equiv M_2^i, 1 \le i \le n$. Hence we have $M_1 \equiv M_2$.

Since M_1 is in β-normal form, we have exhausted all cases. ∎

We now conclude the proof of the theorem. Let $M_1 \equiv \boldsymbol{t\lambda}(\Delta_1), M_2 \equiv \boldsymbol{t\lambda}(\Delta_2)$. Also let $\mathcal{A} \rhd M_1' : \tau, \mathcal{A} \rhd M_2' : \tau$ be normal form terms such that $\mathcal{A} \rhd M_1 \twoheadrightarrow_{T\Lambda} M_1' : \tau$ and $\mathcal{A} \rhd M_2 \twoheadrightarrow_{T\Lambda} M_2' : \tau$ (such M_1', M_2' always exist). By lemma 1, there are normal form terms $\mathcal{A} \rhd e_1 : \tau$ and $\mathcal{A} \rhd e_2 : \tau$ such that $\boldsymbol{er}(M_1') \equiv e_1, \boldsymbol{er}(M_2') \equiv e_2$ and $\mathcal{A} \rhd e \twoheadrightarrow_{\text{ML}} e_1 : \tau$ and $\mathcal{A} \rhd e \twoheadrightarrow_{\text{ML}} e_2 : \tau$. By the uniqueness of normal form, $e_1 \equiv e_2$. Thus $\boldsymbol{er}(M_1') \equiv \boldsymbol{er}(M_2')$. Then by lemma 2, $M_1' \equiv M_2'$. ∎

3.3 Semantics of Core-ML

We define the semantics of Core-ML relative to a model of $T\Lambda$. We first define the semantics of Core-ML typings and then "lift" them to general Core-ML terms.

Let $\mathcal{M}$ be any given model of $T\Lambda$. The semantics of ML typings relative to $\mathcal{M}$ is defined as

$$\mathcal{M}[\![\mathcal{A} \rhd e : \tau]\!]^{\text{ML}}\varepsilon = \mathcal{M}[\![\mathcal{A} \rhd \boldsymbol{t\lambda}(\Delta) : \tau]\!]\varepsilon$$

for some derivation Δ of $\mathcal{A} \rhd e : \tau$. By theorem 6 and the soundness of $T\Lambda$ theories (theorem 4), this definition does not depend on the choice of Δ.

For a given type assignment scheme Σ, the set $TA(\mathcal{A})$ of admissible type assignments under Σ is the set $\{\mathcal{A} | \exists\theta.\ \mathcal{A}\uparrow^{dom(\Sigma)} = \theta(\Sigma)\}$. Under a given type assignment $\mathcal{A}$, the set $TP(\mathcal{A}, \Sigma \rhd e : \rho)$ of types associated with a term $\Sigma \rhd e : \rho$ is the set $\{\tau | \exists\theta.\ (\mathcal{A}\uparrow^{dom(\Sigma)}, \tau) = \theta(\Sigma, \rho)\}$. For a set of types S, we write $\Pi\tau \in S.\ D_\tau$ for the space of functions f such that $dom(f) = S, f(\tau) \in D_\tau$. Then the semantics $\mathcal{M}[\![\Sigma \rhd e : \rho]\!]^{\text{ML}}$ of a Core-ML term $\Sigma \rhd e : \rho$ relative to a model $\mathcal{M}$ is the function taking a type assignment $\mathcal{A} \in TA(\Sigma)$ and an environment $\varepsilon \in Env^{\mathcal{M}}(\mathcal{A})$ that returns an element in $\Pi\tau \in TP(\mathcal{A}, \Sigma \rhd e : \rho).\ D_\tau$ defined as follows:

$$\mathcal{M}[\![\Sigma \rhd e : \rho]\!]^{\text{ML}}\mathcal{A}\varepsilon = \{(\tau, \mathcal{M}[\![\mathcal{A} \rhd e : \tau]\!]^{\text{ML}}\varepsilon) | \tau \in TP(\mathcal{A}, \Sigma \rhd e : \rho)\}$$

For example,

$$\mathcal{M}[\![\emptyset \rhd \lambda x.\ x : t \to t]\!]^{\text{ML}}\mathcal{A}\varepsilon = \{(\tau \to \tau, \mathcal{M}[\![\mathcal{A} \rhd \lambda x : \tau.\ x : \tau \to \tau]\!]\varepsilon) | \tau \in Types\}$$

Now if each element of $D_{\tau_1 \to \tau_2}$ is a function from D_{τ_1} to D_{τ_2} then we have

$$\mathcal{M}[\![\emptyset \rhd \lambda x.x : t \to t]\!]^{\text{ML}}\mathcal{A}\varepsilon = \{(\tau \to \tau, id_{D_\tau}) | \tau \in Types\}$$

where id_X is the identity function on X.

4 Soundness and Completeness of Core-ML Theories

Let $\mathcal{M}$ be a given model of $T\Lambda$. $\mathcal{M}$ also determines the semantics of ML. We say that an equation $\mathcal{A} \rhd e_1 = e_2 : \rho$ is *valid* in $\mathcal{M}$, write $\mathcal{M} \models_{\text{ML}} \Sigma \rhd e_1 = e_2 : \rho$, iff $\mathcal{M}[\![\Sigma \rhd e_1 : \rho]\!]^{\text{ML}} = \mathcal{M}[\![\Sigma \rhd e_2 : \rho]\!]^{\text{ML}}$ (as mappings). Let $\boldsymbol{Valid}^{\text{ML}}(\mathcal{M})$ be the set of all equations in Core-ML that are valid in $\mathcal{M}$. Write $\mathcal{M} \models_{\text{ML}} F$ for $F \subseteq \boldsymbol{Valid}^{\text{ML}}(\mathcal{M})$.

Theorem 7 (Soundness of Core-ML Theories) *Let E_{ML} be any set of ML-equations and $\mathcal{M}$ be any model. If $\mathcal{M} \models_{\text{ML}} E_{\text{ML}}$, then $Th_{\text{ML}}(E_{\text{ML}}) \subseteq \boldsymbol{Valid}^{\text{ML}}(\mathcal{M})$.*

Proof Define mappings Φ, Ψ between sets of ML-equations and sets of $T\Lambda$-equations as:

$$\Phi(E_{\text{ML}}) = \{\mathcal{A} \rhd M_1 = M_2 : \tau | \exists(\mathcal{A} \rhd e_1 = e_2 : \tau) \in E_{\text{ML}} \text{ s.t. } \boldsymbol{er}(M_1) \equiv \mathit{letexpd}(e_1), \boldsymbol{er}(M_2) \equiv \mathit{letexpd}(e_2)\}$$

$$\Psi(E_{T\Lambda}) = \{\Sigma \rhd e_1 = e_2 : \rho | \text{ for any instance } (\mathcal{A}, \tau) \text{ of } (\Sigma, \rho) \ \exists(\mathcal{A} \rhd M_1 = M_2 : \tau) \in E_{T\Lambda} \text{ s.t. } \boldsymbol{er}(M_1) \equiv \mathit{letexpd}(e_1), \boldsymbol{er}(M_2) \equiv \mathit{letexpd}(e_2)\}$$

The proof uses the following lemmas:

Lemma 3 *For any set of ML-equations E_{ML}, $\Psi(Th_{T\Lambda}(\Phi(E_{\text{ML}}))) = Th_{\text{ML}}(E_{\text{ML}})$.*

Proof By our assumptions on E_{ML} and the properties of the rules of ML-theories, $\Sigma \rhd e_1 = e_2 : \rho \in Th_{\mathrm{ML}}(E_{\mathrm{ML}})$ iff for any instance $(\mathcal{A}, \tau)$ of (Σ, ρ), $\mathcal{A} \rhd e_1 = e_2 : \tau \in Th_{\mathrm{ML}}(E_{\mathrm{ML}})$. By definition of Ψ, $\Psi(Th_{T\Lambda}(\Phi(E_{\mathrm{ML}})))$ also has this property. It is therefore enough to show that $\mathcal{A} \rhd e_1 = e_2 : \tau \in Th_{\mathrm{ML}}(E_{\mathrm{ML}})$ iff $\mathcal{A} \rhd e_1 = e_2 : \tau \in \Psi(Th_{T\Lambda}(\Phi(E_{\mathrm{ML}})))$, which is proved by the relationship between the sets of rules of $T\Lambda$ and Core-ML and the definition of Ψ. ▌

Lemma 4 *Let $\mathcal{M}$ be any model.* $\boldsymbol{Valid}^{\mathrm{ML}}(\mathcal{M}) = \Psi(\boldsymbol{Valid}^{T\Lambda}(\mathcal{M}))$.

Proof Suppose $\Sigma \rhd e_1 = e_2 : \rho \in \boldsymbol{Valid}^{\mathrm{ML}}(\mathcal{M})$. For any ground instance $(\mathcal{A}, \tau)$ of (Σ, ρ), $\mathcal{M}[\![\mathcal{A} \rhd e_1 : \tau]\!]^{\mathrm{ML}} = \mathcal{M}[\![\mathcal{A} \rhd e_2 : \tau]\!]^{\mathrm{ML}}$. Let Δ_1, Δ_2 be derivations for $\mathcal{A} \rhd e_1 : \tau$ and $\mathcal{A} \rhd e_2 : \tau$ respectively. Then $\boldsymbol{er}(\boldsymbol{t\lambda}(\Delta_1)) \equiv \mathit{letexpd}(e_1)$, $\boldsymbol{er}(\boldsymbol{t\lambda}(\Delta_2)) \equiv \mathit{letexpd}(e_2)$, and $\mathcal{M}[\![\mathcal{A} \rhd \boldsymbol{t\lambda}(\Delta_1) : \tau]\!] = \mathcal{M}[\![\mathcal{A} \rhd \boldsymbol{t\lambda}(\Delta_2) : \tau]\!]$. Therefore by definition $\Sigma \rhd e_1 = e_2 : \rho \in \Psi(\boldsymbol{Valid}^{T\Lambda}(\mathcal{M}))$. Conversely, suppose $\Sigma \rhd e_1 = e_2 : \rho \in \Psi(\boldsymbol{Valid}^{T\Lambda}(\mathcal{M}))$. Let $(\mathcal{A}, \tau)$ be any instance of (Σ, ρ). By the definition of Ψ, there are M_1, M_2, $\mathcal{A} \rhd M_1 = M_2 : \tau \in \boldsymbol{Valid}^{T\Lambda}(\mathcal{M})$, $\boldsymbol{er}(M_1) \equiv \mathit{letexpd}(e_1), \boldsymbol{er}(M_2) \equiv \mathit{letexpd}(e_2)$. Let Δ_1, Δ_2 be derivations of $\mathcal{A} \rhd e_1 : \tau$ and $\mathcal{A} \rhd e_2 : \tau$ respectively. Then it is shown by using lemmas 1 and 2 that $\mathcal{A} \rhd \boldsymbol{t\lambda}(\Delta_1) =_{T\Lambda} M_1 : \tau$ and $\mathcal{A} \rhd \boldsymbol{t\lambda}(\Delta_2) =_{T\Lambda} M_2 : \tau$. Then by theorem 4, $\mathcal{M}[\![\mathcal{A} \rhd e_1 : \tau]\!]^{\mathrm{ML}} = \mathcal{M}[\![\mathcal{A} \rhd e_2 : \tau]\!]^{\mathrm{ML}}$. Since $(\mathcal{A}, \tau)$ is arbitrary instance of (Σ, ρ), we have $\Sigma \rhd e_1 = e_2 : \rho \in \boldsymbol{Valid}^{\mathrm{ML}}(\mathcal{M})$. ▌

We now conclude the proof of the theorem. Suppose $\mathcal{M} \models_{\mathrm{ML}} E_{\mathrm{ML}}$. By the definitions of Ψ and $\mathcal{M}[\![\,]\!]$, $\mathcal{M} \models_{T\Lambda} \Phi(E_{\mathrm{ML}})$. By theorem 4, $Th_{T\Lambda}(\Phi(E_{\mathrm{ML}})) \subseteq \boldsymbol{Valid}^{T\Lambda}(\mathcal{M})$. Since Ψ is monotone with respect to $\subseteq$, by lemma 3 and 4, $Th_{\mathrm{ML}}(E_{\mathrm{ML}}) \subseteq \boldsymbol{Valid}^{\mathrm{ML}}(\mathcal{M})$. ▌

Theorem 8 *For any set of ML-equations E_{ML} and any model $\mathcal{M}$, if $\boldsymbol{Valid}^{T\Lambda}(\mathcal{M}) = Th_{T\Lambda}(\Phi(E_{\mathrm{ML}}))$ then $\boldsymbol{Valid}^{\mathrm{ML}}(\mathcal{M}) = Th_{\mathrm{ML}}(E_{\mathrm{ML}})$*

Proof By lemma 3 and 4. ▌

Then by theorem 4, we have:

Corollary 1 (Completeness of Core-ML Theories) *For any ML-theory G, there exists a model $\mathcal{G}$ such that $\boldsymbol{Valid}^{\mathrm{ML}}(\mathcal{G}) = G$.* ▌

As a special case of theorem 8, for any model $\mathcal{M}$, we have $\boldsymbol{Valid}^{\mathrm{ML}}(\mathcal{M}) = Th_{\mathrm{ML}}(\emptyset)$ if $\boldsymbol{Valid}^{T\Lambda}(\mathcal{M}) = Th_{T\Lambda}(\emptyset)$. Now let $\mathcal{S}$ be a *full type structure* where D_b is a countably infinite set, $D_{\tau_1 \to \tau_2}$ is the set of all functions from D_{τ_1} to D_{τ_2} and $\bullet$ is the function application. Friedman showed that [Fri73] $\boldsymbol{Valid}^{T\Lambda}(\mathcal{S}) = Th_{T\Lambda}(\emptyset)$. Then we have:

Corollary 2 $\boldsymbol{Valid}^{\mathrm{ML}}(\mathcal{S}) = Th_{\mathrm{ML}}(\emptyset)$

This means that $=_{\mathrm{ML}}$ is sound and complete in the full type structure generated by countably infinite base sets. Since $=_{\mathrm{ML}}$ is decidable, this implies that the set of all true ML equations in the full type structure is recursively enumerable.

5 Extensions of Core-ML

As a programming language, Core-ML should be extended to support recursion and various data types including recursive types. This is done by adding constants and extending the set of types and type-schemes as (possibly infinite) trees generated by various type constructor symbols.

We assume that we are given a ranked alphabet *Tycon* representing a set of type constructors. As observed in [Cop85, Wan84], an appropriate class of infinite trees to support recursive types is the set of *regular trees* [Cou83]. The set of types and the set of type-schemes are extended to the sets of regular trees generated by *Tycon* and $\mathit{Tycon} \cup \mathit{Tvar}$ respectively, where *Tvar* is a set of type variables. As an example, the regular tree represented by the following regular system [Cou83] is a type-scheme of polymorphic lists:

$$L = \mathit{nil} + (t \times L)$$

where $+$ and $\times$ are binary type constructors representing sum and product and *nil* is a trivial type representing the empty list.

We also assume that there is a given set of constant symbols *Const*, each of which is associated with a type-scheme. For example, the products can be introduced by assuming the following set of constants:

$$\begin{aligned} \mathit{pair} &: t_1 \to t_2 \to (t_1 \times t_2) \\ \mathit{first} &: (t_1 \times t_2) \to t_1 \\ \mathit{second} &: (t_1 \times t_2) \to t_2 \end{aligned}$$

The set of raw terms is extended with elements of *Const* (without their associated type-schemes). The proof system for typings is extended by the rule for constants:

(CONST) $\mathcal{A} \rhd c : \tau$ if $\exists (c : \rho) \in \mathit{Consts}$ such that τ is an instance of ρ

We call the extended language ML. The type inference problem of ML is still decidable. Theorem 2 holds also for ML, whose proof uses the unification algorithm for regular trees due to Huet [Hue76].

In order to define a semantics of ML, we need to extend the simply typed lambda calculus $T\Lambda$ and its semantics. The extension of the syntax of $T\Lambda$ is done simply by adding typed constants c^τ and the obvious type-checking axiom for constants. We call the extended

language $T\Lambda^+$. The notion of models is extended by adding a type-preserving interpretation function $\mathcal{C}$ for constants and the condition $[\![\mathcal{A} \rhd c^\tau : \tau]\!]\varepsilon = \mathcal{C}(c^\tau)$ on the semantic mapping. Breazu-Tannen and Meyer extended [BTM85] Friedman's soundness and completeness of equational theories (theorem 4) to languages with constants and a set of types satisfying arbitrary constraints. Since the set of types of $T\Lambda^+$ satisfies their definition of *type algebra*, the soundness and completeness of equational theories still holds for $T\Lambda^+$.

The relationship between $T\Lambda^+$ terms and derivations of ML typings is essentially unchanged and theorem 5 still holds (by adding the case for constants). However, theorem 6 no longer holds. There are non convertible $T\Lambda^+$ terms that correspond to a same ML typing. For example, consider the ML typing:

$$\emptyset \rhd (\mathit{second}((\mathit{pair}\ \lambda x.\, x)1)) : \mathit{int}$$

There are $T\Lambda^+$ terms that correspond to the above ML typing but are not convertible (in $=_{T\Lambda}$). An obvious implication of this fact is that we cannot interpret constants arbitrarily. We need to restrict models of $T\Lambda^+$ to those that give same denotations to terms whenever they correspond to derivations of a same ML typing. The required condition for a model $\mathcal{M}$ of $T\Lambda^+$ is that if $\boldsymbol{er}(M_1) \equiv \boldsymbol{er}(M_2)$ then $\mathcal{M}[\![\mathcal{A} \rhd M_1 : \tau]\!] = \mathcal{M}[\![\mathcal{A} \rhd M_2 : \tau]\!]$. We call a model $\mathcal{M}$ of $T\Lambda^+$ satisfying this condition *abstract*. An abstract model is a model in which the following equations are valid:

$$\mathcal{A} \rhd M_1 = M_2 : \tau \quad \text{if } \boldsymbol{er}(M_1) = \boldsymbol{er}(M_2).$$

By the completeness theorem for equational theories, $T\Lambda^+$ always has an abstract model. We further think that the class of abstract models covers a wide range of standard models of $T\Lambda^+$. For example, ordinary interpretation of *pair* and *second* certainly satisfies the above condition and suggests an abstract model. Any abstract model of $T\Lambda^+$ yields a semantics of ML. The soundness and completeness of equational theories of ML (theorem 7 and 8) hold with respect to the class of abstract models. Proofs are same as before except that they use the condition of abstract models in place of theorem 6. The condition of abstract models can be regarded as a necessary condition for *fully abstract models* we will exploit in the next section.

6 Full Abstraction of ML

One desired property of a denotational semantics of a programming language is *full abstraction* [Mil77, Plo77, Mul84, MC88], which roughly says that the denotational semantics coincides with the operational semantics. In this section, we will show that if a model of $T\Lambda^+$ is fully abstract for an operational semantics of $T\Lambda^+$ then it is also fully abstract for the corresponding operational semantics of ML.

Following [Plo77, MC88], we define an operational semantics as a partial function on closed terms of *base types*. Let $\mathcal{E}^{T\Lambda}, \mathcal{E}^{\mathrm{ML}}$ be respectively the evaluation functions of $T\Lambda^+$ and ML determining their operational semantics. We write $\mathcal{E}(X) \Downarrow y$ to mean that $\mathcal{E}(X)$ is defined and equal to y. On the operational semantics of $T\Lambda^+$ we assume that it depend only on structure of terms. Formally, we assume $\mathcal{E}^{T\Lambda}$ to satisfy the following property:

> for two terms $\emptyset \rhd M_1 : b$ and $\emptyset \rhd M_2 : b$ if $\boldsymbol{er}(M_1) \equiv \boldsymbol{er}(M_2)$ then $\mathcal{E}^{T\Lambda}(\emptyset \rhd M_1 : b) \Downarrow \emptyset \rhd c^b : b$ iff $\mathcal{E}^{T\Lambda}(\emptyset \rhd M_2 : b) \Downarrow \emptyset \rhd c^b : b$.

We believe this condition to be satisfied by most operational semantics of explicitly-typed programming languages. On the operational semantics of $\mathcal{E}^{\mathrm{ML}}$ we assume the following property on evaluation of ***let***-expressions:

> $\mathcal{E}^{\mathrm{ML}}(\emptyset \rhd e : b) \Downarrow \emptyset \rhd c : b$ iff $\mathcal{E}^{\mathrm{ML}}(\emptyset \rhd \mathit{letexpd}(e) : b) \Downarrow \emptyset \rhd c : b$.

This condition correspond to the equality axiom (let). Note that the rule (let) corresponds to the rule (β) and does not agree with the call-by-value evaluation strategy. Finally we assume the following relationship between the operational semantics of $T\Lambda^+$ and that of ML:

> for terms $\emptyset \rhd M : b$ of $T\Lambda^+$ and $\emptyset \rhd e : b$ of ML, if $\boldsymbol{er}(M) \equiv e$ then $\mathcal{E}^{T\Lambda}(\emptyset \rhd M : b) \Downarrow \emptyset \rhd c^b : b$ iff $\mathcal{E}^{\mathrm{ML}}(\emptyset \rhd e : b) \Downarrow \emptyset \rhd c : b$.

We believe that in most cases it is routine to construct $\mathcal{E}^{\mathrm{ML}}$ from given $\mathcal{E}^{T\Lambda}$ that satisfies this condition and vice versa.

A context $C[\ ]$ in $T\Lambda^+$ is a $T\Lambda^+$ pre-term with one "hole" in it. We omit a formal definition. A context $C[\ ]$ is a *closing b-context for* $\mathcal{A} \rhd M : \tau$ if there is a derivation for $\boldsymbol{T\Lambda} \vdash \emptyset \rhd C[M] : b$ such that its subderivation for (the occurrence in $C[M]$ of) M is a derivation for $\mathcal{A} \rhd M : \tau$. Two $T\Lambda^+$ terms $\mathcal{A} \rhd M : \tau$ and $\mathcal{A} \rhd N : \tau$ are *operationally equivalent*, write $\mathcal{A} \rhd M \overset{T\Lambda}{\approx} N : \tau$, iff for any closing b-context $C[\ ]$ for these two terms, $\mathcal{E}^{T\Lambda}(\emptyset \rhd C[M] : b) \Downarrow \emptyset \rhd c^b : b$ iff $\mathcal{E}^{T\Lambda}(\emptyset \rhd C[N] : b) \Downarrow \emptyset \rhd c^b : b$.

In ML, under our assumption on ***let***-expressions, it is enough to consider raw terms and contexts that do not contain ***let***-expression. We therefore define a context $c[\ ]$ in ML as a context of the untyped lambda calculus. A context $c[\ ]$ is a *closing b-context for* $\Sigma \rhd e : \rho$ if there is a derivation for $\emptyset \rhd c[e] : b$ such that its subderivation for e is a derivation for an instance of $\Sigma \rhd e : \rho$. Two ML terms $\Sigma \rhd e_1 : \rho$ and $\Sigma \rhd e_2 : \rho$ are *operationally equivalent*, write $\Sigma \rhd e_1 \overset{\mathrm{ML}}{\approx} e_2 : \rho$, iff for any closing

b-context $c[\ ]$ for these two terms, $\mathcal{E}^{\mathrm{ML}}(\emptyset \rhd c[e_1] : b) \Downarrow \emptyset \rhd c : b$ iff $\mathcal{E}^{\mathrm{ML}}(\emptyset \rhd c[e_2] : b) \Downarrow \emptyset \rhd c : b$.

A model $\mathcal{M}$ is *fully abstract for* $\mathcal{E}^{T\Lambda}$ if $\mathcal{M} \models_{T\Lambda} \mathcal{A} \rhd M = N : \tau$ iff $\mathcal{A} \rhd M \overset{T\Lambda}{\approx} N : \tau$. Similarly, a mode $\mathcal{M}$ is *fully abstract for* $\mathcal{E}^{\mathrm{ML}}$ if $\mathcal{M} \models_{\mathrm{ML}} \Sigma \rhd e_1 = e_2 : \rho$ iff $\Sigma \rhd e_1 \overset{\mathrm{ML}}{\approx} e_2 : \rho$. Note that a model $\mathcal{M}$ is fully abstract for $\mathcal{E}^{T\Lambda}$ then it is an abstract model. This means that any fully abstract model for $\mathcal{E}^{T\Lambda}$ yields a semantics of ML. Moreover, we have:

Theorem 9 *If a model $\mathcal{M}$ is fully abstract for $\mathcal{E}^{T\Lambda}$ then $\mathcal{M}$ is also fully abstract for $\mathcal{E}^{\mathrm{ML}}$.*

Proof Let $\mathcal{M}$ be any fully abstract model for $\mathcal{E}^{T\Lambda}$. By our assumption on $\mathcal{E}^{\mathrm{ML}}$ and the definition of $\mathcal{M}[\![\ ]\!]^{\mathrm{ML}}$, it is sufficient to show the condition of full abstraction for $\mathcal{E}^{\mathrm{ML}}$ for terms that do not contain ***let***-expression. Suppose $\Sigma \rhd e_1 \overset{\mathrm{ML}}{\approx} e_2 : \rho$, where e_1, e_2 do not contain ***let***-expression. By the definition of $\overset{\mathrm{ML}}{\approx}$, $\mathcal{A} \rhd e_1 \overset{\mathrm{ML}}{\approx} e_2 : \tau$ for any ground instance $(\mathcal{A}, \tau)$ of (Σ, ρ). Let $\mathcal{A} \rhd M : \tau, \mathcal{A} \rhd N : \tau$ be $T\Lambda^+$ terms such that $\boldsymbol{er}(M) \equiv e_1, \boldsymbol{er}(N) \equiv e_2$. By using the assumption on the relationship between $\overset{T\Lambda}{\approx}$ and $\overset{\mathrm{ML}}{\approx}$, it is easily shown that $\mathcal{A} \rhd M \overset{T\Lambda}{\approx} N : \tau$. By the full abstraction of $\mathcal{M}$ for $\mathcal{E}^{T\Lambda}$, by theorem 5 and by definition of $\mathcal{M}[\![\ ]\!]^{\mathrm{ML}}$, $\mathcal{M} \models_{\mathrm{ML}} \Sigma \rhd e_1 = e_2 : \rho$. Conversely, suppose $\mathcal{M} \models_{\mathrm{ML}} \Sigma \rhd e_1 = e_2 : \rho$, where e_1, e_2 do not contain ***let***-expression. Let $c[\ ]$ be any closing b-context for $\Sigma \rhd e_1 : \rho$ and $\Sigma \rhd e_1 : \rho$. Let Δ_1 be a derivation for $\emptyset \rhd c[e_1] : b$ such that it contains a subderivation Δ_2 for $\mathcal{A} \rhd e_1 : \tau$ where $(\mathcal{A}, \tau)$ is an instance of (Σ, ρ). Since $c[\,]$ is a closing b-context for $\Sigma \rhd e_1 : \rho$, such Δ_1 always exists. Since $\Sigma \rhd e_2 : \rho$ is a term, there is a derivation tree Δ_3 for $\mathcal{A} \rhd e_2 : \tau$. Then by typing rules and the definition of contexts, the derivation tree Δ_4 obtained from Δ_1 by replacing the subtree Δ_2 by Δ_3 is a derivation for $\emptyset \rhd c[e_2] : b$. Let $M_1 \equiv \boldsymbol{t\lambda}(\Delta_1)$, $M_2 \equiv \boldsymbol{t\lambda}(\Delta_2)$, $M_3 \equiv \boldsymbol{t\lambda}(\Delta_3)$, $M_4 \equiv \boldsymbol{t\lambda}(\Delta_4)$. Then $\boldsymbol{er}(M_2) \equiv e_1$, $\boldsymbol{er}(M_3) \equiv e_2$. Clearly M_1, M_4 respectively contain M_2, M_3 as subterms. Moreover, the $T\Lambda^+$ contexts obtained from M_1, M_4 by replacing M_2, M_3 with the 'hole' are identical. Call this context $C[\ ]$. Then $\boldsymbol{er}(C[M_2]) \equiv c[e_1]$ and $\boldsymbol{er}(C[M_3]) \equiv c[e_2]$, Since $\mathcal{M} \models_{\mathrm{ML}} \Sigma \rhd e_1 = e_2 : \rho$, $\mathcal{M} \models_{T\Lambda} \mathcal{A} \rhd M_2 = M_3 : \tau$. Then by the full abstraction of $\mathcal{M}$ for $\mathcal{E}^{T\Lambda}$, $\mathcal{E}^{T\Lambda}(\rhd C[M_2] : \tau) \Downarrow \rhd c^b : b$ iff $\mathcal{E}^{T\Lambda}(\rhd C[M_3] : \tau) \Downarrow \rhd c^b : b$. Then by the assumption on the relationship between $\overset{T\Lambda}{\approx}$ and $\overset{\mathrm{ML}}{\approx}$, $\mathcal{E}^{\mathrm{ML}}(\emptyset \rhd c[e_1] : b) \Downarrow \emptyset \rhd c : b$ iff $\mathcal{E}^{\mathrm{ML}}(\emptyset \rhd c[e_2] : b) \Downarrow \emptyset \rhd c : b$. ∎

The importance of this result is that we can immediately apply results already developed for explicitly typed languages to implicitly typed language with ML polymorphism. As an example, in [Plo77] Plotkin constructed a fully abstract model of his language PCF with parallel conditionals. It is not hard to define the "ML version" of PCF (with parallel conditionals) by deleting type specifications of bound variables and adding let expressions. Its operational semantics can be also defined in such a way that it satisfies our assumptions. We then immediately have a fully abstract model for the ML-version of PCF.

7 Conclusion

We have presented a framework for denotational semantics and equational theories for ML. We have characterized a term of ML as a pair of a raw term and a type-scheme and defined its denotation as a set of pairs of a simple type and a value in a semantic space of the typed lambda calculus. We have axiomatized the equational theories of ML and proved their soundness and completeness with respect to our semantics. Based on this semantic framework, we have also presented a result that allows us to transfer existing full abstraction results from typed lambda calculi to ML-like languages.

One limitation of our framework is that it assumes β-equality both in equational theories and in semantic spaces (through a condition on semantic mappings). Because of this fact our framework does not fit "call-by-value" evaluation. Since the "call-by-value" evaluation is also widely used, it would be useful to develop a framework for denotational semantics and equational theories that precisely capture the behavior of ML programs under the "call-by-value" evaluation.

Acknowledgement

I would like to thank Val Breazu-Tannen and Peter Buneman for discussions and many useful suggestions.

References

[Aug84] L. Augustsson. A compiler for lazy ML. In *Symposium on LISP and Functional Programming*, pages 218–227. ACM, 1984.

[BTM85] V. Breazu-Tannen and A. R. Meyer. Lambda calculus with constrained types (extended abstract). In R. Parikh, editor, *Proceedings of the Conference on Logics of Programs, Brooklyn, June 1985*, pages 23–40. *Lecture Notes in Computer Science*, Vol. 193, Springer-Verlag, 1985.

[Cop84] M. Coppo. Completeness of type assignment in continuous lambda models. *Theoretical Computer Science*, 29:309–324, 1984.

[Cop85] M. Coppo. A completeness theorem for recursively defined types. In G. Goos and J. Hartmanis, editors, *Automata, Languages and Programming, 12th Colloquium, LNCS 194*, pages 120–129. Splinger-Verlag, July 1985.

[Cou83] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.

[DM82] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[Fri73] H. Friedman. Equations between functionals. In *Lecture Notes in Mathematics 453*, pages 22–33. Springer-Verlag, 1973.

[Hin69] R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. American Mathematical Society*, 146:29–60, December 1969.

[Hin83] R. Hindley. The completeness theorem for typing λ-terms. *Theoretical Computer Science*, 22:1–17, 1983.

[HMM86] R. Harper, D. B. MacQueen, and R. Milner. Standard ML. LFCS Report Series ECS-LFCS-86-2, Department of Computer Science, University of Edinburgh, March 1986.

[HS86] J. R. Hindley and J. P. Seldin. *Introduction to Combinators and λ-Calculus*. Cambridge University Press, 1986.

[Hue76] G Huet. *Résolution d'équations dans les langages d'ordre 1,2,...ω*. PhD thesis, University Paris, 1976.

[MC88] A.R. Meyer and S.S. Cosmadakis. Semantical paradigms. In *Proc. IEEE Symposium on Logic in Computer Science*, July 1988.

[MH88] J. C. Mitchell and R. Harper. The essence of ML. pages 28–46, San Diego, California, January 1988.

[Mil77] R. Milner. Fully abstract models of typed λ-calculi. *Theoretical Computer Science*, 4:1–22, 1977.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[MPS86] D.B. MacQueen, G.D. Plotkin, and Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71(1/2):95–130, 1986.

[Mul84] K. Mulmuley. A semantic characterization of full abstraction for typed lambda calculus. In *Proc. 25-th IEEE Symposium on Fundations of Computer Science*, pages 279–288, 1984.

[Oho89] A. Ohori. *A Study of Types, Semantics and Languages for Databases and Object-oriented Programming*. PhD thesis, University of Pennsylvania, 1989.

[Plo77] G.D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[Tur85] D.A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science 201*, pages 1–16. Springer-Verlag, 1985.

[Wan84] M. Wand. A types-as-sets semantics for Milner-style polymorphism. pages 158–164, January 1984.