Compilation Based on a Calculus for Explicit Type Passing

Yasuhiko Minamide
Research Institute for Mathematical Sciences, Kyoto University
Kyoto, 606-01, JAPAN
E-mail: nan@kurims.kyoto-u.ac.jp

ABSTRACT

We propose several calculi for explicit type passing that enable us to formalize compilation of polymorphic programming languages like ML as phases of type-preserving translations. In our calculi various manipulations for type parameters can be expressed without typing problems—this is impossible in the polymorphic λ-calculi. Furthermore, we develop the translation from an explicit typed source calculus similar to Core-XML to one of the proposed calculi which completely eliminates runtime construction type parameters. We propose an intermediate language based on this calculus, and discuss an implementation of a compiler for Core Standard ML.

1. Introduction

One of the advantages of typed programming languages is that type information can be productively used to obtain efficient executable programs. However, in the presence of polymorphism, a type in a program may contain type variables and cannot be determined as a ground type at compile time. This has been one of the main difficulties in developing implementations of ML which effectively utilize type information obtained from type inference. In order to overcome this problem two approaches to utilizing type information for compilation have been proposed recently.

The first approach is representation analysis proposed by Leroy [13], which is based on coercions between monomorphic and polymorphic types. Although the polymorphic parts of a program must use uniform representation, the monomorphic parts can be compiled efficiently by using type information. It was reported that the compilers based on this approach improved the performance of executable programs [13, 21].

More aggressive approach is to use type information by passing types as actual parameters [17, 9, 4, 26, 25] as in the second order λ-calculus. The advantage of this approach is that even for a polymorphic type the actual type is known as an instance at runtime and can be used for several purposes. For example, Tolmach implemented tag-free garbage collection by passing type parameters [26]. Harper and Morrisett proposed a general mechanism for using types at runtime called intensional type analysis [9]. In Tolmach's implementation it is shown that explicit type passing is not so expensive as one might think.

In this paper, we propose several calculi for explicit type passing that enable us to formalize compilation of polymorphic programming languages like ML as phases of type-preserving translations. In our calculi, various manipulations for type parameters can be expressed without typing problems, which is impossible in standard

polymorphic λ-calculi. Furthermore, we propose the translation from a source calculus similar to Core-XML to one of the proposed calculi which completely eliminates runtime construction of type parameters. Based on the calculus, we propose an intermediate language and discuss an implementation of an experimental compiler for Core Standard ML [15].

There are many advantages of compilers which are constructed as phases of type-preserving translations. Complete type information can be used for optimization such as instantiating polymorphic equality to monomorphic equality and choosing efficient representation of data types. It is also useful to prove correctness of compilation through such a method as logical relations [24, 6, 20, 22, 23]. Furthermore, constructing compilers as phases of type-preserving translations has a practical advantage for development of compilers as mentioned in [16, 25]. When we debug a compiler itself, a code of an intermediate language can be type-checked. This has greatly helped to find bugs in our compiler.

Type parameter passing is sometimes considered expensive. For example, Tolmach reported that the memory allocated for type parameters sometimes exceeded the memory saved by tag-free garbage collection. Thus, it is desirable to consider optimization of manipulation of type parameters. We propose the transformation that eliminates runtime construction of type parameters. However, it is found that the transformation cannot be formulated as a type-preserving translation in standard polymorphic λ-calculi. Thus, we develop a more suitable calculus to express manipulations of type parameters.

There have been many studies that considered passing some kind of type information to implement various aspects of programming languages such as polymorphic record operations and overloading [27, 18, 11]. For the theoretical basis of these approaches, various typed languages have been proposed. The calculus $\Lambda^{\Rightarrow}$ we propose in this paper can be considered as an extension of the implementation calculus for compilation of polymorphic records of Ohori [18, 19] and qualified types of Jones [12, 11]. Runtime type parameters are distinguished from the usual types and new abstraction and application for runtime type parameters are introduced. In our calculus, various manipulations of types are more naturally expressed.

The next step is to develop the translation from the source language to the calculus $\Lambda^{\Rightarrow}$ where type parameter passing is made explicit. There are many ways to achieve this depending on what type information is to be passed to each polymorphic function. The simplest choice is to pass the type information for type variables as Tolmach's. This strategy gives complete type information and enables tag-free garbage collection. However, if we only want the type information for some non-parametric operations, only the type information for relevant type variables need to be passed. Conversely, to avoid runtime construction of type parameters, more complicated type information is passed to a type function. Thus, we formulate a deductive system that non-deterministically chooses what type information is passed to each polymorphic

function. It is proved that this general translation preserves the type and operational behaviour of a program. As a special case of the deductive system we propose a translation called *aggressive lifting*, where the program obtained through the translation requires no runtime construction of type parameters. Furthermore, we formulate deterministic deductive systems for Tolmach's and aggressive lifting. These systems naturally give rise to the algorithms of both types of lifting.

We propose another calculus $\lambda^{\Rightarrow}$ which identifies type parameters with values. It is shown that $\lambda^{\Rightarrow}$ has greater expressiveness on manipulations of type parameters and is more suited to an actual intermediate language of compilers. Since type parameters can be considered as values, they can be components of tuples. By using this facility, closures can be implemented more efficiently. Furthermore, optimization such as uncurrying of type parameter and usual abstractions is more simply formulated than in the previous calculus.

We have implemented an experimental compiler by using Kit compiler [2] as front end. The compiler translates Core Standard ML programs to C. Non-parametric operations such as polymorphic equality and tag-free garbage collection are implemented by using runtime type parameters. As the intermediate language of the compiler, we adopted the language based on calculus $\lambda^{\Rightarrow}$ proposed in this paper. Our implementation of the compiler is such that Tolmach's and aggressive type lifting can be chosen at compile time. The effect of this choice is measured on the simple benchmarks like polymorphic equality and standard benchmarks such as `life` and `knuth-bendix`. The results for simple benchmarks show that aggressive type lifting is more than 15 % faster at execution time than Tolmach's. For standard benchmarks, the advantage of aggressive lifting varies from the minimal to 10 % at execution time and 35 % at heap allocation depending on benchmark programs.

Recently, there have been several attempts to utilize type information of ML programs by passing type information. There are two compilers of ML that pass type information at runtime: Tolmach's compiler [26] and TIL [25]. The intermediate language of Tolmach's compiler seems basically untyped. On the other hand, similarly as our compiler TIL compiler is organized as phases of type-preserving translations. However, TIL compiler does not consider optimizations of type parameter passing and their typing problems. The intermediate language of the compiler is based on the second order λ-calculus and intensional type analysis.

2. Explicit Type Parameters

There are no explicit type abstractions and applications in ML. However, as proposed in [8], ML programs can be translated to the language with explicit type abstractions and applications like the second order λ-calculus. Milner's type inference algorithm finds a typing derivation for a well-typed program in the type system. Based on the derivation, we can translate ML programs to the explicitly typed lan-

guage as described in [8]. For example, the following program is translated into the explicitly-typed program below.

```
let fun f x =
    let fun g y = (x,y)
    in
        (g 2, g true)
    end
in
    f 3; f 3.14
end
```

```
let val f = Λt. λx:t.
    let val g = Λs. λy:s. (x,y)
    in
        (g {int} 2, g {bool} true)
    end
in
    f {int} 3; f {real} 3.14
end
```

However, in order to preserve the operational behavior of a program we have to restrict generalization of types in `let` for values. Such restriction was studied in [28] to avoid imperative types and reported to work well in practice.

Thus, as the source language of compilation of ML we consider the restricted polymorphic λ-calculus similar to the Core-XML [8] that captures typing properties of Core Standard ML. The source language is defined as follows:

$$
\begin{array}{llll}
\textit{Monotypes} & \tau & ::= & b \mid t \mid \tau \to \tau \\
\textit{Polytypes} & \sigma & ::= & \tau \mid \forall t_1, \ldots, t_n.\tau \\
\textit{Expressions} & e & ::= & c \mid x \mid \lambda x{:}\tau.e \mid e_1 e_2 \mid \Lambda t_1, \ldots, t_n.e \mid e\{\tau_1, \ldots, \tau_n\} \mid \mathsf{p} \mid \\
 & & & \quad \texttt{let}\ x{:}\sigma = e_1\ \texttt{in}\ e_2
\end{array}
$$

We call this calculus $\lambda^{\forall}$. For the illustration of compilation of non-parametric operations, the primitive p of type $\forall t.\tau_{\mathsf{p}}$ is included in this calculus. We assume that primitive p requires some type information at runtime. We often write a sequence of types as $\overline{\tau}$. For example, we sometimes write a type application as $e\{\overline{\tau}\}$. For this calculus, we consider the standard type system and call-by-value operational semantics.

As source programs of compilation, we consider only the type normalized expressions [10] that satisfy the following two conditions. Type abstractions occur only as the bound expression of `let`-expressions; i.e., `let` $x{:}\sigma = \Lambda t_1, \ldots, t_n.e_1$ `in` e_2. Type applications are only allowed for variables; i.e, $x\{\tau_1, \ldots, \tau_n\}$. The type normalized expression corresponding to an expression can be obtained by some simple transformations without changing behaviour of the expression.

2.1. Tolmach's Lifting Transformation

There are several possibilities on how type information can be passed in programs. The simplest way is to treat each type variable as a variable for values and implement type substitution as substitution for values. However, such implementation seems to introduce a lot of cost.

Thus, Tolmach proposed passing several type parameters as one parameter that he called a type environment and performing type substitution lazily. To pass type parameters as one parameter, a program is transformed so that every type function is closed with respect to type variables. In the above example, we have type function `g` with free type variable t. (Although type variable t does not occur in function `g`, it should be considered free because variable `x` of type t occurs in function `g`.) To make `g` closed with respect to type variables, free type variable t is lifted and t is abstracted with s in an uncurried abstraction as below.

```
let val g = Λts.λx:t.λy:s. (x,y)
    val f = Λt.λx:t.(g {t,int} x 2, g {t,bool} x true)
in
    f {int} 3; f {real} 3.14
end
```

We call this transformation Tolmach's lifting. This transformation is always possible in $\lambda^{\forall}$ because all the application sites of a type function can be determined syntactically in this language. Then type arguments for an uncurried type abstraction are passed as one parameter. Thus, only one parameter is necessary for type information in a type function after applying this transformation.

It should be remarked that variable `x` is also lifted in function `g`, because the type of `x` is t that is lifted. Without such lifting, the transformation may result in an ill-typed program. Thus, the transfromation depends on which variables should be considered free in a type function. This information is usually determined during closure conversion to avoid closures for known functions [1]. Thus, this lifting transformation and closure conversion must be performed at the same time. That makes the implementation of a compiler based on this transformation compilicated.

It is still expensive to construct a type environment every time a type function is called. Thus, Tolmach used the method performing type substitution lazily. In

his implementation, almost all parts of a type enviornment are created statically and each construction of type environment consumes only 2 words.

2.2. Aggressive Lifting and Typing Problems

In this subsection we consider optimized type parameter passing and show that it is impossible to express such type parameter passing in standard polymorphic λ-calculi. The following example is taken from [26]. At runtime of the program, the type environment $\{s \times s\}$ for `pair` is created every time the function `repair` is called.

```
let pair = Λt.λx:t.(x,x)
    repair = Λs.λy:s. pair {s × s} (y,y)
in
    repair {int} n
end
```

To overcome this inefficiency, Tolmach suggested lifting the type parameter for the function `pair` out of `repair` as below.

```
let pair  = Λt.λx:t.(x,x)
    repair = Λs.Λt.λy:s. pair {t} (y,y)
in
    repair {int} {int×int} n
end
```

Now that the type paramter {`int`×`int`} is closed, that can be constructed statically and there are no need of runtime construction of types. If a calling site of `repair` is in another type function, the lifted type argument may be still open. Thus, in order to eliminate runtime construction of type parameters we must apply this transformation from the inner-most type functions repeatedly. Thus, if there are nested calls to type fucntions, the number of type arguments may increase exponentially.

Furthermore, this program is not well-typed in the second order λ-calculus because t is not equal to $s \times s$ in the function `repair`. One solution of this problem is to use *translucent types* (or *manifest types*) proposed for modules of Standard ML [7, 14] and recently used to solve the typing problem of closure conversion of the second order λ-calculus in [16]. For the program above, we can add the constraint $t = s \times s$ at abstraction of t in `repair` and such modification makes the program well-typed in the extended calculus. However, this solution is not satisfactory when we consider other kinds of manipluation of types as we see below.

In order to avoid the increase of the number of type parameters, the transformation is modified so that two type absractions are uncurried as below.

```
let pair = Λt.λx:t.(x,x)
    repair = Λs.λy:π1(s). pair {π2(s)}.(y,y)
in
    repair {⟨int,int×int⟩} n
end
```

where uncurrying is made explicit by using type abstraction that takes a pair of types $\langle \tau_1, \tau_2 \rangle$ instead of uncurried form of type abstraction $\Lambda t_1, t_2.e$. We call this transformation *aggressive lifting* in this paper. In this transformation, the number of the type arguments does not increase and the runtime construction of types are replaced by projections of type parameters. However, even if we use translucent types for this transformation, it is impossible to make this program well-typed. The constraint that the type argument s of `repair` must hold is $\pi_2(s) = \pi_1(s) \times \pi_1(s)$ and we cannot express this constraint by translucent types. So, we propose a calculus more suitable to express manupulations of type parameters in the next section.

3. A Calculus for Explicit Type Passing

In this section, we propose the explicitly typed calculus that is suitable to express wide range of manipulations of type parameters and thus the base of the intermediate language of compilers of ML. There have been many studies that proposed passing some kind of type information to implement various aspects of programming languages such as polymorphic record operations and overloading [27, 18, 11]. For the theoretical basis of these approaches, various calculi have been proposed. The calculus we propose can be considered as an extension of the implementation calculus for compilation of polymorphic records of Ohori [18, 19] and qualified types of Jones [12, 11]. We mainly use terminology of qualified types in this section. The language of our calculus is defined as follows:

$$
\begin{array}{llll}
\textit{Monotypes} & \tau & ::= & b \mid t \mid \tau \to \tau \mid \phi \Rightarrow \tau \\
\textit{Predicates} & \phi & ::= & \tau \mid \langle \phi_1, \ldots, \phi_n \rangle \\
\textit{Polytypes} & \sigma & ::= & \tau \mid \forall t_1, \ldots, t_n.\tau \\
\textit{Evidences} & \omega & ::= & b \mid u \mid \omega \to \omega \mid \pi_i(\omega) \mid \langle \omega_1, \ldots, \omega_n \rangle \\
\textit{Expressions} & e & ::= & c \mid x \mid \lambda x{:}\tau.e \mid e_1 e_2 \mid \Lambda t_1, \ldots, t_n.e \mid e\{\tau_1, \ldots, \tau_n\} \mid \\
 & & & \Lambda u{:}\phi.e \mid e[\omega] \mid \mathtt{p} \mid \mathtt{let}\ x{:}\sigma = e_1\ \mathtt{in}\ e_2
\end{array}
$$

Compared to $\lambda^{\forall}$, the language is extended by predicates and evidences. Evidences are corresponding to runtime type parameters for our purpose and are classified by predicates. Monotypes are extended by type $\phi \Rightarrow \tau$ which is the type for the function that takes a type parameter classified by ϕ. Evidences (or runtime type parameters) have the definition similar to that of monotypes to express corresponding type information. However, a type variable is replaced by an evidence variable u and it is extended by tuple $\langle \omega_1, \ldots, \omega_n \rangle$ and projection $\pi_i(\omega)$, which are used to pass multiple

$$\text{(base)}\quad \Delta;\Sigma \Vdash b{:}b \qquad \text{(var)}\quad \frac{u{:}\phi \in \Sigma}{\Delta;\Sigma \Vdash u{:}\phi} \qquad \text{(arrow)}\quad \frac{\Delta;\Sigma \Vdash \omega_1{:}\tau_1 \quad \Delta;\Sigma \Vdash \omega_2{:}\tau_2}{\Delta;\Sigma \Vdash \omega_1 \to \omega_2{:}\tau_1 \to \tau_2}$$

$$\text{(tuple)}\quad \frac{\Delta;\Sigma \Vdash \omega_i{:}\phi_i \quad (1 \le i \le n)}{\Delta;\Sigma \Vdash \langle \omega_1,\ldots,\omega_n\rangle{:}\langle \phi_1,\ldots,\phi_n\rangle} \qquad \text{(proj)}\quad \frac{\Delta;\Sigma \Vdash \omega{:}\langle \phi_1,\ldots,\phi_n\rangle}{\Delta;\Sigma \Vdash \pi_i(\omega){:}\phi_i}$$

Figure 1: Rules for Evidences

types as one parameter. The abstraction $\Lambda u{:}\phi.e$ represents the function which takes type information corresponding to ϕ. The runtime type information ω is given to an evidence abstraction by evidence application $e[\omega]$. We call this calculus $\Lambda^{\Rightarrow}$, where Λ is evidence abstraction and $\Rightarrow$ is the type of evidence abstraction.

Now we will give the type system for the calculus. First, we need the following three kinds of contexts.

Type contexts	Δ	::=	$t_1,\ldots,t_n$
Predicate assignments	Σ	::=	$u_1{:}\phi_1,\ldots,u_n{:}\phi_n$
Type assignments	Γ	::=	$x_1{:}\tau_1,\ldots,x_n{:}\tau_n$

We extend the usage of type contexts to general sequence of type variables. For example, we sometimes write polytypes and type abstractions as $\forall\Delta.\tau$ and $\Lambda\Delta.e$. There are two kinds of judgments as follows:

$\Delta;\Sigma \Vdash \omega{:}\phi$	evidence ω satisfies predicate ϕ
$\Delta;\Sigma;\Gamma \vdash e{:}\tau$	expression e has type τ

where we assume that all free type variables in Σ and Γ are included in Δ. The rules for judgment $\Delta;\Sigma \Vdash \omega{:}\phi$ are shown in Figure 1. Intuitively, $\Delta;u_1{:}\phi_1,\ldots,u_n{:}\phi_n \Vdash \omega{:}\phi$ means that evidence ω is the type information corresponding to predicate ϕ if we assume each u_i is the type information corresponding to predicate ϕ_i. According to our specific purpose, these rules are simplified compared to the rules of qualified types. However, they are extended for tuples and projections of evidences.

The typing rules for the calculus are similar to the rules of qualified types and are shown in Figure 2. Only the rules for type abstraction and application, and evidence abstraction and application are shown. Other rules are standard. For primitive p of type $\forall t.\tau_{\mathsf{p}}$ in $\lambda^{\forall}$, we assume that the corresponding primitive p in $\Lambda^{\Rightarrow}$ requires the evidence (or type parameter) which satisfies predicate ϕ_{p} and thus has type $\forall t.\phi_{\mathsf{p}} \Rightarrow \tau_{\mathsf{p}}$ in $\Lambda^{\Rightarrow}$.

For this calculus, we consider the standard call-by-value operational semantics:

$$\frac{\Delta;\Sigma,u{:}\phi;\Gamma \vdash e:\tau}{\Delta;\Sigma;\Gamma \vdash \Lambda u{:}\phi.e:\phi \Rightarrow \tau} \qquad \frac{\Delta;\Sigma;\Gamma \vdash e:\phi \Rightarrow \tau \quad \Delta;\Sigma \Vdash \omega:\phi}{\Delta;\Sigma;\Gamma \vdash e[\omega]:\tau}$$

$$\frac{\Delta,\Delta';\Sigma;\Gamma \vdash e:\tau}{\Delta;\Sigma;\Gamma \vdash \Lambda\Delta'.e:\forall\Delta'.\tau} \qquad \frac{\Delta;\Sigma;\Gamma \vdash e:\forall t_1,\ldots,t_n.\tau}{\Delta;\Sigma;\Gamma \vdash e\{\tau_1,\ldots,\tau_n\}:\tau[\tau_1,\ldots,\tau_n/t_1,\ldots,t_n]}$$

$$\Delta;\Sigma;\Gamma \vdash \mathsf{p}:\forall t.\phi_{\mathsf{p}} \Rightarrow \tau_{\mathsf{p}}$$

Figure 2: Typing Rules

$(\Lambda u{:}\phi.e)[\omega]$ is reduced to $e[\omega/u]$.

3.1. Representing Type Passing in $\Lambda^{\Rightarrow}$

In this subsection, we consider how various ways of type lifting we have considered can be represented in this calculus without causing typing problems. The formal account of the translation from the source language $\lambda^{\forall}$ to the calculus $\Lambda^{\Rightarrow}$ is given in Section 4.

The following program is the translation of the first example corresponding to Tolmach's lifting.

```
let val f = Λt.Λu:⟨t⟩.λx:t.
    let val g = Λs.Λu:⟨t,s⟩.λy:s. (x,y)
    in
        (g {int} [⟨t,int⟩] 2, g {bool} [⟨t,bool⟩] true)
    end
in
    f {int} [⟨int⟩] 3; f {real} [⟨real⟩] 3.14
end
```

For the function `f` and `g`, the evidence abstractions are added after the type abstractions. The evidence parameter `u` for `g` takes an evidence corresponding type variables t and s. Thus, all the necessary type information can be obtained from one parameter `u` in the function `g`. It can be easily checked that this program is well-typed in $\Lambda^{\Rightarrow}$.

Furthermore, as the example shows, we do not have to lift type and value abstractions. Thus we can separate the lifting transformation from closure conversion and the design of compilers can be simplified. Furthermore, this representation is more closely reflecting Tolmach's implementation since it is explicitly expressed that all runtime type arguments to a type function are passed as one argument.

Furthermore, we can express various manipulations of type parameters without causing the typing problems we described in Section 2.2. In the following example, the type parameter for the function `pair` is lifted from the function `repair` as type parameter $\mathtt{u}_2$.

$$
\begin{array}{l}
\mathtt{let\ pair} = \Lambda t.\,\Lambda \mathtt{u}{:}\langle t\rangle.\,\lambda \mathtt{x}{:}t.\,(\mathtt{x},\mathtt{x}) \\
\quad \mathtt{repair} = \Lambda s.\,\Lambda \mathtt{u}_1{:}\langle s\rangle.\,\Lambda \mathtt{u}_2{:}\langle s\times s\rangle.\,\lambda \mathtt{y}{:}s.\ \mathtt{pair}\ \{s\times s\}\ [\mathtt{u}_2]\ (\mathtt{y},\mathtt{y}) \\
\mathtt{in} \\
\quad \mathtt{repair}\ \{\mathtt{int}\}\ [\langle \mathit{int}\rangle]\ [\langle \mathit{int}\times \mathit{int}\rangle]\ 1 \\
\mathtt{end}
\end{array}
$$

The function `pair` has type $\forall t.\langle t\rangle \Rightarrow t \to t\times t$ and `pair` $\{s\times s\}$ has type $\langle s\times s\rangle \Rightarrow (s\times s)\to (s\times s)\times(s\times s)$. Since $\mathtt{u}_2$ satisfies predicate $\langle s\times s\rangle$ and `(y,y)` has type $s\times s$, `pair` $\{s\times s\}$ `[u2] (y,y)` is well-typed.

In the same way, uncurrying of type parameters can be done while preserving well-typedness of the program as the following. The type parameter consiting of s and $\langle s\times s\rangle$ is passed as one parameter to `repair` and its second component is used as the type parameter to the funciton `pair`.

$$
\begin{array}{l}
\mathtt{let\ pair} = \Lambda t.\,\Lambda \mathtt{u}{:}\langle t\rangle.\,\lambda \mathtt{x}{:}t.\,(\mathtt{x},\mathtt{x}) \\
\quad \mathtt{repair} = \Lambda s.\,\Lambda \mathtt{u}{:}\langle s,\langle s\times s\rangle\rangle.\,\lambda \mathtt{y}{:}s.\ \mathtt{pair}\ \{s\times s\}\ [\pi_2(\mathtt{u})]\ (\mathtt{y},\mathtt{y}) \\
\mathtt{in} \\
\quad \mathtt{repair}\ \{\mathtt{int}\}\ [\langle \mathit{int},\langle \mathit{int}\times \mathit{int}\rangle\rangle]\ 1 \\
\mathtt{end}
\end{array}
$$

4. Type Lifting Translation

In this section, we will give the translation from source calculus $\lambda^{\forall}$ to calculus $\Lambda^{\Rightarrow}$. First, we consider a general translation that is non-deterministic and involves not only Tolmach's lifting but also aggressive lifting. The translation is given as $\Delta;\Sigma;\Gamma \vdash e \rightsquigarrow e'$ where Δ, Σ, and Γ are a type context, a predicate assignment, and a type assignment of $\Lambda^{\Rightarrow}$ respectively. The closed expressions are translated under the empty contexts.

The rules for the translation are shown in Figure 3. In rule (let), the body of the type abstraction, e_1, is translated by assuming that type information is passed by fresh evidence variable u. This variable u is assumed to hold the evidence that satisfies predicate ϕ. However, no condition is given for the predicate ϕ in this rule. That makes the translation non-deterministic and gives the ability to express various ways of lifting. As we see later, several restrictions are imposed in this rule depending on the strategies of lifting. After the translation of e_1, e_2 is translated by assuming that x takes an evidence that satisfies predicate ϕ.

A type application is translated into the combination of type and evidence applications by rule (tapp). The applied type parameter is determined from the type

(let)

$$\frac{\Delta, \Delta'; \Sigma, u{:}\phi; \Gamma \vdash e_1 \rightsquigarrow e_1' \quad \Delta; \Sigma; \Gamma, x{:}\forall\Delta'.\phi \Rightarrow \tau \vdash e_2 \rightsquigarrow e_2'}{\Delta; \Sigma; \Gamma \vdash \texttt{let}\ x{:}\forall\Delta'.\tau = \Lambda\Delta'.e_1\ \texttt{in}\ e_2 \rightsquigarrow \texttt{let}\ x{:}\forall\Delta'.\phi \Rightarrow \tau = \Lambda\Delta'.\Lambda u{:}\phi.e_1'\ \texttt{in}\ e_2'}$$

(tapp)

$$\frac{\Delta; \Sigma \Vdash \omega : \phi_1[\overline{\tau}/\Delta']}{\Delta; \Sigma; \Gamma, x{:}\forall\Delta'.\phi_1 \Rightarrow \tau_2 \vdash x\{\overline{\tau}\} \rightsquigarrow x\{\overline{\tau}\}[\omega]}$$

(prim)

$$\frac{\Delta; \Sigma \Vdash \omega : \phi_{\mathsf{p}}[\tau/t]}{\Delta; \Sigma; \Gamma \vdash \mathsf{p}\{\tau\} \rightsquigarrow \mathsf{p}\{\tau\}[\omega]}$$

Figure 3: Translation from $\lambda^{\forall}$ to $\Lambda^{\Rightarrow}$

assignment of the variable. If the variable with type $\forall t_1, \ldots, t_n.\phi_1 \Rightarrow \tau_2$ is applied to the type $\tau_1, \ldots, \tau_n$ then it requires the type parameter corresponding to $\phi_1[\tau_1, \ldots, \tau_n/t_1, \ldots, t_n]$. So the evidence ω that satisfies $\phi_1[\tau_1, \ldots, \tau_n/t_1, \ldots, t_n]$ under Σ is applied. For primitive p, $\mathsf{p}\{\tau\}$ is applied to the evidence ω which satisfies $\phi_{\mathsf{p}}[\tau/t]$.

For this deductive system, we proved that the translation preserves the type and operational behaviour of a program. Operational correctness is proved by the method of logical relations.

4.1. Tolmach's Lifting

Tolmach's lifting is formulated by restricting the rule (let) as follows:

$$\frac{\Delta, \Delta'; \Sigma, u{:}\langle\Delta, \Delta'\rangle; \Gamma \vdash e_1 \rightsquigarrow e_1' \qquad \Delta; \Sigma; \Gamma, x{:}\forall\Delta'.\langle\Delta, \Delta'\rangle \Rightarrow \tau \vdash e_2 \rightsquigarrow e_2'}{\begin{array}{l}\Delta; \Sigma; \Gamma \vdash \texttt{let}\ x{:}\forall\Delta'.\tau = \Lambda\Delta'.e_1\ \texttt{in}\ e_2 \\ \qquad\qquad \rightsquigarrow \texttt{let}\ x{:}\forall\Delta'.\langle\Delta, \Delta'\rangle \Rightarrow \tau = \Lambda\Delta'.\Lambda u{:}\langle\Delta, \Delta'\rangle.e_1'\ \texttt{in}\ e_2'\end{array}}$$

where the predicate ϕ is restricted to Δ, Δ'. This choice of predicate Δ, Δ' for u is always enough to translate e_1. By this restriction the translation is almost deterministic. However, the deduction of $\Vdash$ is still non-deterministic and may give redundant translation such as $\Vdash \pi_1(\langle b \rangle) : b$. Thus, Tolmach's algorithm is obtained by restricting the (var) and (proj) rules as follows:

$$\Delta; \Sigma, u{:}\langle t_1, \ldots, t_n\rangle \Vdash \pi_i(u){:}t_i.$$

4.2. Aggressive Lifting

In order to avoid runtime construction of types, we restrict evidence ω as follows:

$$\omega ::= \omega' \mid u \mid \pi_i(\omega)$$

where ω' is a closed evidence that does not contain projection π_i. A translation obtained with this restriction does not have to construct type parameters at runtime, since there is no evidence variable within $\langle \ldots \rangle$ and $\rightarrow$. However, the restricted deductive system does not provide the algorithm for the translation because it is not known at the translation of `let` what open evidences are necessary to translate e_1 and thus what predicate ϕ should be used for translation of e_1. Thus, we will define the deterministic version of the deductive system that provides a translation algorithm.

The deterministic version of the deductive system is obtained by collecting necessary types during translation. The relation has the following form: $\Delta; u; \Gamma \vdash e; \langle \phi_1, \ldots, \phi_m \rangle \rightsquigarrow e'; \langle \phi_1, \ldots, \phi_m, \ldots, \phi_{m+k} \rangle$, where u is the current evidence variable for obtaining type information. The predicates which u must satisfy are created during the translation of e. First we assume that u must hold the evidences for $\phi_1, \ldots, \phi_m$. Then, the predicates necessary to translate e, $\phi_{m+1} \ldots \phi_{m+k}$, are appended to the list. The rules of the translation are given below.

$$\frac{\phi[\overline{\tau}/\Delta'] \text{ is open.}}{\Delta; u; \Gamma, x{:}\forall\Delta'.\phi \Rightarrow \tau_2 \vdash x\{\overline{\tau}\}; \langle \phi_1, \ldots, \phi_n \rangle \rightsquigarrow x\{\overline{\tau}\}[\pi_{n+1}(u)]; \langle \phi_1, \ldots, \phi_n, \phi[\overline{\tau}/\Delta'] \rangle}$$

$$\frac{\phi[\overline{\tau}/\Delta'] \text{ is closed.} \quad \emptyset; \emptyset \Vdash \omega : \phi[\overline{\tau}/\Delta']}{\Delta; \Sigma; \Gamma, x : \forall\Delta'.\phi \Rightarrow \tau_2 \vdash x\{\overline{\tau}\}; \phi \rightsquigarrow x\{\overline{\tau}\}[\omega]; \phi}$$

$$\frac{\Delta, \Delta'; u; \Gamma \vdash e_1; \langle \Delta, \Delta' \rangle \rightsquigarrow e_1'; \phi_0 \qquad \Delta; u'; \Gamma, x{:}\forall\Delta'.\phi_0 \Rightarrow \tau \vdash e_2; \phi' \rightsquigarrow e_2'; \phi''}{\begin{array}{l}\Delta; u'; \Gamma \vdash \texttt{let}\ x{:}\forall\Delta'.\tau = \Lambda\Delta'.e_1\ \texttt{in}\ e_2; \phi' \\ \qquad\qquad \rightsquigarrow \texttt{let}\ x{:}\forall\Delta'.\phi \Rightarrow \tau = \Lambda\Delta'.\Lambda u{:}\phi.e_1'\ \texttt{in}\ e_2'; \phi''\end{array}}$$

In case of the first rule, since predicate $\phi[\overline{\tau}/\Delta']$ is open, if we create the evidence here, the runtime construction of the evidence is necessary. Instead, we add predicate $\phi[\overline{\tau}/\Delta']$ to the list of the predicates and apply $x\{\overline{\tau}\}$ to the projection of current evidence variable u. However, since in the second rule $\phi[\overline{\tau}/\Delta']$ is closed the evidence to corresponding to it can be created statically. In the rule for `let`, expression e_1 is translated with $\langle \Delta, \Delta' \rangle$. That guarantees that the type information for every free type variable can be obtained from u and tag-free garbage collection can be performed by using type information obtained from u.

The following lemma ensures that there is a deduction in the original system corresponding to a deduction of the algorithmic version of the deductive system.

Lemma 1 *If* $\Delta; u; \Gamma \vdash e; \phi \rightsquigarrow e'; \phi'$, *then* $\Delta; u{:}\phi'; \Gamma \vdash e \rightsquigarrow e'$.

Polytype $\forall\Delta'.\phi \Rightarrow \tau$ in $\Lambda^{\Rightarrow}$ is called a translation of $\forall\Delta'.\tau$ under Δ if $FTV(\phi) \subseteq \Delta \cup \Delta'$. This relation is naturally extended to type assignments.

Lemma 2 *Let* Γ' *be a translation of* Γ *under* Δ. *If* $\Delta; \Gamma \vdash e : \tau_0$, *then for all* ϕ *there exist* e' *and* ϕ' *such that* $\Delta; u; \Gamma' \vdash e; \phi \rightsquigarrow e'; \phi'$.

It is clear from this lemma that for $\emptyset;\emptyset \vdash e : \tau$ there exists a deduction of $\emptyset;u;\emptyset \vdash e;\langle\rangle \rightsquigarrow e';\langle\rangle$. This ensures the existence of a translation for an arbitrary well-typed closed program.

5. Identifying Evidences with Values

In this section, we propose a calculus which is more expressive for manipulation of evidences and is more suitable for an intermediate language of a compiler. In calculus $\Lambda^{\Rightarrow}$, there are three kinds of abstractions and applications for types, evidences, and values. Abstractions and applications for types are completely ignored at runtime, though enable type checking of programs. However, the latter two are implemented in the similar way. Thus, by identifying the latter two we obtain the following calculus:

$$
\begin{array}{llll}
\textit{Monotypes} & \tau & ::= & b \mid t \mid \tau \to \tau \mid T(\phi) \mid \langle \tau_1 \times \ldots \times \tau_n \rangle \\
\textit{Predicate} & \phi & ::= & \tau \mid \langle \phi_1, \ldots, \phi_n \rangle \\
\textit{Polytypes} & \sigma & ::= & \tau \mid \forall \Delta.\tau \\
\textit{Evidence} & \omega & ::= & b \mid E(e) \mid \omega \to \omega \mid \pi_i(\omega) \mid \langle \omega_1, \ldots, \omega_n \rangle \\
\textit{Expressions} & e & ::= & c \mid x \mid \lambda x{:}\tau.e \mid e_1 e_2 \mid \Lambda\Delta.e \mid e\{\tau_1, \ldots, \tau_n\} \mid \mathtt{p} \mid \\
 & & & R(\omega) \mid \langle e_1, \ldots, e_n \rangle \mid \pi_i(e) \mid \mathtt{let}\ x{:}\sigma = e_1\ \mathtt{in}\ e_2
\end{array}
$$

where $T(\phi)$ is the type for the evidences satisfying predicate ϕ, $E(e)$ coerces expressions to evidences, and $R(\omega)$ coerces evidences to expressions. Since evidences are abstracted by usual λ we call this calculus $\lambda^{\Rightarrow}$. In order to illustrate advantages of $\lambda^{\Rightarrow}$ over the previous calculus, the tuple of expressions $\langle e_1, \ldots, e_n \rangle$ and its type $\langle \tau_1 \times \ldots \times \tau_n \rangle$ are included in the calculus. As the evidences may contain free variables the judgments of the calculus has the following forms:

$$
\begin{array}{ll}
\Delta;\Gamma \Vdash \omega : \phi & \text{evidence } \omega \text{ satisfies predicate } \phi \\
\Delta;\Gamma \vdash e : \tau & \text{expression } e \text{ has type } \tau
\end{array}
$$

The rules to identify evidences with values are defined as follows:

$$
\frac{\Delta;\Gamma \vdash e : T(\phi)}{\Delta;\Gamma \Vdash E(e) : \phi} \qquad \frac{\Delta;\Gamma \Vdash \omega : \phi}{\Delta;\Gamma \vdash R(\omega) : T(\phi)}
$$

Other rules are analogous to those of $\Lambda^{\Rightarrow}$.

This calculus is more expressive in manipulation of evidences (or type parameters). For example, since type parameters can now be considered as values they can be components of tuples. Furthermore, it is possible to express a function that returns a type parameter as bellow.

$$
\lambda x{:}T(t).R(E(x) \to E(x)) : T(t) \to T(t \to t)
$$

This function takes the evidence corresponding to type t and returns the evidence corresponding to $t \to t$. Optimization such as uncurrying abstractions for a type

parameter and a value is formulated more simply than in the previous calculus. Thus, we think that this calculus is more suitable for an intermediate language of a compiler.

Now we consider the representation of closures in both calculi. Since evidences are distinguished from usual values in $\Lambda^{\Rightarrow}$, an evidence cannot be a component of a usual record. Thus, the evidence and value environments of a closure cannot be merged into a single environment. For example, the closure of $\lambda x{:}t.y$ for $y = 1$ and $t = int$ is represented in $\Lambda^{\Rightarrow}$ as follows:

$$\langle(\Lambda t.\Lambda u{:}t.\lambda y{:}t.\lambda x{:}t.y)\{int\}, int, 1\rangle$$

In order to keep uniform representation for closures we cannot eliminate the evidence abstraction in the code and evidence component of a closure even if there are no free evidence variables. Thus the closure of $\lambda x{:}int.x+y$ for $y = 1$ is represented as follows:

$$\langle\Lambda u{:}\langle\ \rangle.\lambda y{:}int.\lambda x{:}int.x + y, \langle\ \rangle, 1\rangle$$

where the evidence environment is empty $\langle\ \rangle$. Furthermore, we need a special constructor for closures since it has an evidence as a component.

However, in $\lambda^{\Rightarrow}$ an evidence can be a component of a record of expressions. Hence, the evidence and value environments can be merged into a single environment. The closures of $\lambda x{:}t.y$ and $\lambda x{:}int.x + y$ are represented as follows:

$$\langle(\Lambda t.\lambda z{:}\langle T(t) \times t\rangle.\lambda x{:}t.\pi_2(z))\{int\}, \langle R(int), 1\rangle\rangle$$

$$\langle\lambda y{:}int.\lambda x{:}int.x + y, 1\rangle$$

As we see in the example above, this representation saves space for empty evidence environments in closures.

Even if we use $\lambda^{\Rightarrow}$, we need a special constructor for closures to hide the type of the environment of a closure. However, as described in [16], we can represent a closure of $\lambda^{\Rightarrow}$ by a standard tuple and a package expression of existential types. A closure of type $\tau_1 \to \tau_2$ can be represented as a value of type $\exists t.(t \to \tau_1 \to \tau_2) \times t$ where t is the type of the environment.

6. Implementation

An experimental compiler is implemented as a translator from Core SML to C by using Kit Compiler as front end. An executable program produced by the compiler uses type parameters for tag-free garbage collection and non-parametric operations such as polymorphic equality.

The compiler is organized as in Figure 4. The Kit Compiler translates Core SML to the explicitly typed intermediate language Lambda that is similar to $\lambda^{\forall}$. We have designed another intermediate language called IL. It is based on calculus $\lambda^{\Rightarrow}$ proposed

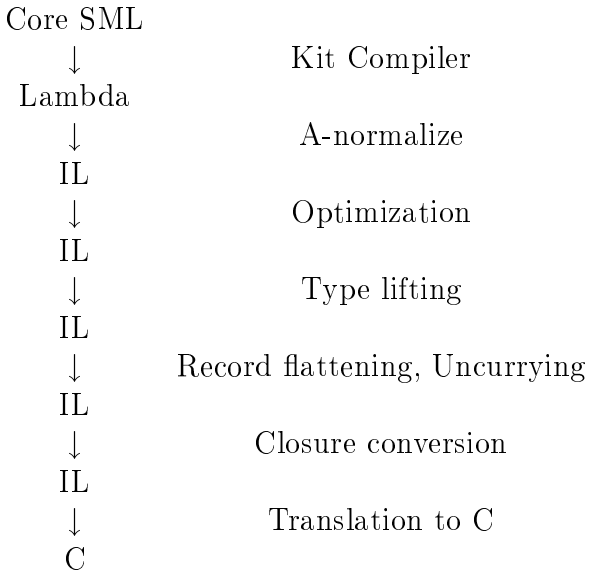

Figure 4: The organization of the compiler

in this paper and on A-normal forms [5, 25]. Since $\lambda^{\forall}$ can be considered as a subset of $\lambda^{\Rightarrow}$, IL is used not only after type lifting but before type lifting. Then, all the stages except the last translation to C are implemented as type-preserving translations.

The type-checker for IL is implemented so that the intermediate code produced by each translation can be type-checked during debugging of the compiler. Type-checking of intermediate code has greatly helped development of our compiler.

We implemented the type lifting translation so that Tolmach's and aggressive lifting can be chosen when we compile a program. Evidences (or runtime type parameters) are implemented by the data structures similar to Tolmach's. For aggressive lifting, the representation of evidences is simplified because only projections of evidences are necessary at runtime. Type parameters are passed to functions as usual C arguments.

Uncurrying optimization is performed after type lifting to eliminate the abstractions and applications introduced by type lifting if possible. Closures are represented by a special constructor as the abstract closures described in [16]. However, closures do not consists of code, type environment, and value environment but just code and environment as described in Section 5.

The polymorphic equality function of SML is implemented as C function which uses runtime type information. In Tolmach's representation, the function must perform computation corresponding to reductions of projections of evidences. However, for aggressive lifting the type arguments to the polymorphic equality function are also lifted and then such reductions are not necessary at all. That simplifies the implementation of polymorphic equality in aggressive lifting and makes the use of type

information in aggressive lifting cheaper than in Tolmach's as described in the next section.

7. Measurements

We measured the cost of passing and using type parameters in aggressive and Tolmach's lifting. In general, when a type parameter corresponding to an open type is passed, Tolmach's lifting constructs a new type environment and allocates 2 words and aggressive lifting has to perform a projection on a type parameter. We first compared the cost of these operations.

Function `repair` in Section 2.2 allocates 4 words for usual pairs. Thus, if we compile this program by instantiating `pair` and `repair` to monomorphic functions, this program consumes 4 words for usual data structures for pairs. This is same for aggressive lifting. However, Tolmach's lifting allocates 2 words for type parameters and thus allocates 1.5 times memory in the whole. An even worse situation for Tolmach's lifting is the following program.

```
let proj = Λt1t2.λx:t1 × t2.π1(x)
    pair-proj = Λs.λy:s. proj {s,s} (y,y)
in
    pair-proj {int} n
end
```

Record flattening optimization of our compiler converts the function `proj` from the function that takes a pair into the function with two arguments. So if we compile this program by instantiating `proj` and `pair-proj` to monomorphic functions or aggressive lifting, this program allocates no words. However, the executable program compiled by Tolmach's lifting allocates 2 words for each call of `pair-proj`. So, even if we considered heap allocation of programs asymptotically, the program compiled by Tolmach's lifting may show unexpected behaviour.

Execution time of these programs are shown in Figure 5. The column Mono shows the results when the programs are compiled by instantiating polymorphic functions to monomorphic functions. The numbers in the parentheses are the ratio to the cases of aggressive lifting. It shows that aggressive lifting is more than 15 % faster than Tolmach's. If we compare aggressive lifting to the monomorphic cases, we cannot see any significant difference in performance.

In order to measure the cost using type information, we measured the execution time of programs using the polymorphic equality function. In Tolmach's representation, the computation corresponding normalizing type parameters must be performed because of lazy reduction of types. The test programs `polyeq1` and `polyeq2` repeatedly applies polymorphic equality to integers, `polyeq2pair` to pairs of integers. Before calling polymorphic equality, there are one nested call of polymorphic function

	Aggressive	Tolmach	Mono
`repair`	10.40	12.33 (1.19)	10.36 (1.00)
`pair-proj`	5.31	7.70 (1.45)	5.30 (1.00)
`polyeq1`	6.43	9.23 (1.44)	-
`polyeq2`	6.43	10.91 (1.70)	-
`polyeq2pair`	16.53	21.85 (1.32)	-

Figure 5: Basic benchmarks: execution time excluding garbage collection time (sec)

in `polyeq1` and two netsted calls in `polyeq2` and `polyeq2pair`. The results are also shown in Figure 5. In both cases aggressive lifting is faster than Tolmach's. In case of aggressive lifting, the type argument to the polymorphic equality is identical for `polyeq1` and `polyeq2` and their execution time are almost the same. On the other hand, if compiled by Tolmach's lifting, there are one more nesting of type environment in case of `polyeq2` and the difference in Tolmach's and our representation in `polyeq2` is bigger than that of `polyeq1`. However, since checking equality is much expensive for `polyeq2pair`, the relative difference of the two is smaller.

We measured the allocation and execution time of standard benchmarks for both aggressive and Tolmach's lifting. The rusults are shown in Figure 6. The results for SML/NJ 0.93 are also shown to show the excutable programs are fast enough to consider the effect of the strategies of type lifting.[a]

The excutable programs compiled by aggressive lifting are always faster than those by Tolmach's except for `mandelbrot`, which is compiled to a completely monomporhic program. The biggest difference about 10 % in execution time is seen for `life`. As reported in [21], `life` spends almost all the execution time for polymorphic equality. (We did not implement minimal typing [3].) Thus, the difference comes from the time spent for polymorphic equality. This seems also true for `boyer` since there is no difference in allocation.

As for allocation the differences between Tolmach's and aggressive lifting are minimal except for `simple`. For `simple`, the executable program compiled by Tolmach's lifting allocates 35 % more memory. This seems to happen because most polymorphic functions are inlined in the other programs. Thus, much bigger difference might be seen when we consider separate compilation.

8. Conclusion

We have proposed several calculi for explicit type passing that enable us to formalize compilation of polymorphic programming languages like ML as phases of type

[a] It is not fair to compare allocation of excutable programs of our compiler to that of SML/NJ because ours uses stack for frames of functions, but SML/NJ uses heap.

	Aggressive		Tolmach's		SML/NJ 0.93	
	Time	Allocation	Time	Allocation	Time	Allocation
`life`	7.78	3.79	8.55 (1.10)	3.92 (1.03)	12.04	39.63
`knuth-bendix`	8.63	16.78	8.82 (1.02)	16.81 (1.00)	11.72	62.88
`simple`	21.75	20.43	22.71 (1.04)	27.61 (1.35)	31.45	153.85
`boyer`	2.48	3.42	2.58 (1.04)	3.42 (1.00)	1.70	9.52
`mandelbrot`	7.83	0.00	7.82 (1.00)	0.00 (1.00)	10.65	46.48

Figure 6: Results for standard benchmarks: time is the execution time (sec) excluding garbage collection time and allocation is in M words.

preserving translations. Furthermore, we have formulated a translation from the source language to the proposed calculus as a non-deterministic deductive system and proved its correctness. The translation algorithm that avoids runtime construction of type parameters has been given as a special case of the deductive system.

Based on the calculus, we have designed an intermediate language and implemented a compiler of Core Standard ML. The results for simple benchmarks show programs compiled by aggressive type lifting are more than 15 % faster at execution time than those compiled by Tolmach's. For standard benchmarks, the advantage is much smaller in general. This seems to happen because most polymorphic functions are inlined. Thus, we would like to extend our compiler to separate compilation and study various strategies of type passing in presence of separate compilation.

Acknowledgements

Some of the ideas presented in this paper were developed while the author was visiting Carnegie Mellon University. We would like to thank Robert Harper and Greg Morrisett for many interesting discussions on the subject of this paper. Irek Ulidowski and the anonymous referees provided helpful comments to improve this paper.

References

[1] Andrew W. Appel. *Compiling with Continuation.* Cambridge University Press, 1992.

[2] Lars Birkedal, Nick Rothwell, Madds Tofte, and David N. Turner. *The ML Kit Version 1*, 1993.

[3] Nikolaj Skallerud Bjorner. Minimal typing derivations. In *ACM SIGPLAN Workshop on ML and its Application*, 1994.

[4] Ctherine Dubois and Pierre Weis. Generic polymorphism. In *Proc. ACM Symp. on Principles of Prog. Languages*, 1995.

[5] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 1993.

[6] H. Friedman. Equality between functionals. In R. Parikh, editor, *Logic Colloquium '75*. North-Holland, 1975.

[7] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules. In *Proc. ACM Symp. on Principles of Prog. Languages*, pages 123–137, 1994.

[8] Robert Harper and John C. Mitchell. On the type structure of standard ML. *ACM Transaction on Programming Languages and Systems*, 15(2), 1993.

[9] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Proc. ACM Symp. on Principles of Prog. Languages*, pages 130–141, 1995.

[10] Fritz Henglein and Jesper Jorgensen. Formally optimal boxing. In *Proc. ACM Symp. on Principles of Prog. Languages*, 1994.

[11] Mark P. Jones. A theory of qualified types. In *ESOP '92: European Symposium on Programming*, LNCS 582, 1992.

[12] Mark P. Jones. ML typing, explicit polymorphism and qualified types. In *TACS '94: Conference on theoretical aspects of computer software*, LNCS 789, 1994.

[13] Xavier Leroy. Unboxed objects and polymorphic typing. In *Proc. ACM Symp. on Principles of Prog. Languages*, 1992.

[14] Xavier Leroy. Manifest types, modules, and separate compilation. In *Proc. ACM Symp. on Principles of Prog. Languages*, pages 109–122, 1994.

[15] Robin Milner, Mads Tafte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[16] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *Proc. ACM Symp. on Principles of Prog. Languages*, 1996.

[17] R. Morrison, A. Dearle, R.C.H. Connor, and A. L. Brown. An ad hoc approach to the implementation of polymorphism. *ACM Transaction on Programming Languages and Systems*, 13(3), 1991.

[18] Atsuhi Ohori. A compilation method for ML-style polymorphic record calculi. In *Proc. ACM Symp. on Principles of Prog. Languages*, 1992.

[19] Atsushi Ohori. A polymorphic record calculus and its compilation. *ACM Transaction on Programming Languages and Systems*, 17(6), 1995.

[20] Gordon D. Plotkin. Lambda-definability in the full type hierarchy. In *To H.B.Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism.* Academic Press, 1980.

[21] Zhong Shao and Andrew W. Appel. A type-based compiler for Standard ML. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 1995.

[22] R. Statman. Completeness, invariance, and lambda-definability. *Journal of Symbolic Logic*, 47:17–26, 1982.

[23] R. Statman. Logical relations and the typed λ-calculus. *Information and Control*, 65, 1985.

[24] W. W. Tait. Intensional interpretation of functionals of finite type. *Journal of Symbolic Logic*, 32(2), 1967.

[25] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P Lee. TIL: A type-directed optimizing compiler for ML. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 1996.

[26] Andrew Tolmach. Tag-free garbage collection using explicit type parameters. In *Proc. ACM Conf. Lisp and Functional Programming*, pages 1–11, 1994.

[27] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proc. ACM Symp. on Principles of Prog. Languages*, 1989.

[28] Andrew K. Wright. Polymorphism for imperative languages without imperative types. Technical report, Rice University, 1993. TR93-200.