Typed Closure Conversion*

Yasuhiko Minamide †
Research Institute for Mathematical Sciences
Kyoto University
Kyoto 606–01, Japan
nan@kurims.kyoto-u.ac.jp

Greg Morrisett
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213–3891
jgmorris@cs.cmu.edu

Robert Harper
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213–3891
rwh@cs.cmu.edu

Abstract

Closure conversion is a program transformation used by compilers to separate code from data. Previous accounts of closure conversion use only *untyped* target languages. Recent studies show that translating to *typed* target languages is a useful methodology for building compilers, because a compiler can use the types to implement efficient data representations, calling conventions, and tag-free garbage collection. Furthermore, type-based translations facilitate security and debugging through automatic type checking, as well as correctness arguments through the method of logical relations.

We present closure conversion as a type-directed, and type-preserving translation for both the simply-typed and the polymorphic λ-calculus. Our translations are based on a simple "closures as objects" principle: higher-order functions are viewed as objects consisting of a single method (the code) and a single instance variable (the environment). In the simply-typed case, the Pierce-Turner model of object typing where objects are packages of existential type suffices. In the polymorphic case, more careful tracking of type sharing is required. We exploit a variant of the Harper-Lillibridge "translucent type" formalism to characterize the types of polymorphic closures.

1 Introduction

Closure conversion [29, 34, 5, 17, 16, 1, 39, 9] is a program transformation that achieves a separation between code and data. Functions with free variables are replaced by code abstracted on an extra environment parameter. Free variables in the body of the function are replaced by references to the environment. The abstracted code is partially applied to an explicitly constructed environment providing the bindings for these variables. This partial application of the code to its environment is in fact suspended until the function is actually applied to its argument; the suspended application, called a *closure*, is a data structure consisting of a piece of pure code and a representation of its environment.

A critical decision in closure conversion is the choice of representation for the environment — whether to use a flat FAM-like [4], linked CAM-like [5], or hybrid representation [30]. The choice of representation is influenced by a desire to minimize closure creation time, the space consumed by the environment, and the time to access a variable in the environment [39, 30]. An important property of closure conversion is that the representation of the environment is *private* to the closure. This affords considerable flexibility in the representation of environments and is thus exploited to good advantage by Shao and Appel [30] and Wand and Steckler [39].

Previous accounts consider closure conversion as a transformation to *untyped* terms, even if the source language is typed [34, 17, 1, 39, 9]. This is adequate for compilers that make little or no use of types in the back end or at run time. However, when compiling typed languages, it is often advantageous to propagate type information through each stage of the compiler, and to make use of types at link- or even run time. For example, Leroy's representation analysis [18, 31] uses types to determine procedure calling conventions, and Ohori's record compilation [25] uses a representation of types at run time to access components of a record. Compilation strategies for polymorphic languages, such as those proposed by Morrison *et al.* [24] and Harper and Morrisett [14], rely on analyzing types at run time to support unboxed representations and non-parametric operators, including printing and structural equality. Tag-free garbage collection [3, 37, 23] for both monomorphic and polymorphic programming languages relies on analyzing types at run time to determine the size and layout of objects in the heap. To support any of these implementation strategies, it is necessary to propagate type information through closure conversion and into the generated code. The purpose of this paper is to demonstrate how this can be done in both a simply-typed and a polymorphic setting.

We present closure conversion as an example of a *type-directed* and *type-preserving* translation. In general, such

*This research was sponsored in part by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software", ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050, and in part by the National Science Foundation under Grant No. CCR-9502674, and in part by the Isaac Newton Institute for Mathematical Sciences, Cambridge, England. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of ARPA or the U.S. Government. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

†This research was performed while the first author was visiting the Fox Project at Carnegie Mellon University.

translations transform both a term and its type, possibly relying on type information to guide the translation. Thus each stage of the compiler can be viewed as a type-preserving translation between typed intermediate languages. Examples of such translations have been given by Leroy [18], Ohori [25], Harper and Lillibridge [10], and Harper and Morrisett [14]. In contrast to type-free compilation strategies, these translations make essential use of type information during translation to increase the time or space efficiency of programs. In addition to the practical advantages of this approach, type-directed translation also facilitates the work of the compiler writer. In particular, the typing properties of the intermediate code may be exploited to give clear, concise proofs of compiler correctness through the method of *logical relations* [35, 6, 27, 32, 33]. Furthermore, the intermediate code of the compiler can be mechanically typed-checked, an important debugging tool for the compiler writer. Finally, checkable typed intermediate languages are a promising technique for ensuring safety properties of programs in a distributed environment [8, 38].

We describe closure conversion for the simply-typed λ-calculus and the predicative fragment of the polymorphic λ-calculus. In each case we present closure conversion in two stages. The first stage, called *abstract closure conversion*, is a type-directed translation to an intermediate language with a primitive notion of closures. We describe the translation as a deductive system where the choice of environment representations may be independently made for each closure. We argue that various representations considered in the literature (such as the FAM [4] or CAM [5]), as well as hybrid representations [30], can all be explained in this uniform framework. We establish the correctness of the translation once for all environment representations.

The second stage, called *closure representation*, is another type-directed translation where closures are implemented in terms of generic typed λ-calculus primitives. The main idea is to represent *closures as objects* consisting of a single method (the code) and a single instance variable (the environment). We show that, in the simply-typed case, Pierce and Turner's type discipline for object-oriented programming [26] may be used to characterize the types of closures. In particular, we use existential type abstraction to ensure the privacy of environment representation in much the same way that Pierce and Turner hide the representation types of instance variables. In the polymorphic case, we must use a more sophisticated type discipline in order to track critical type sharing relationships within the closure. To this end, we exploit a variant of the *translucent type* [11] (or *manifest type* [19]) formalism. Our "closures as objects" model provides an interesting counterpoint to the more familiar "objects as closures" proposal introduced by Reddy [28].

We prove the correctness of both the abstract closure conversion and the closure representation stages using the method of logical relations. The main idea is to define a type-indexed family of simulation relations that establish a correspondence between the source and target terms of the translation. Once a suitable system of relations has been defined, it is relatively straightforward to prove by induction on the definition of the compilation relation that the source and target of the translation are related. From this, we may conclude that a closed program and its translation evaluate to the same result. Due to lack of space, we omit the proofs of correctness here. However, full details may be found in the companion technical report [21].

Closure conversion is discussed in descriptions of various functional language compilers [34, 17, 2, 1, 30]. It is similar to λ-lifting [15] in that it eliminates free variables in the bodies of λ-abstractions but differs by making the representation of the environment explicit as a data structure. Making the environment explicit is important because it exposes environment construction and variable lookup to an optimizer. Furthermore, Shao and Appel show that not all environment representations are "safe for space" [30], and thus choosing a good environment representation is an important part of compilation. Wand and Steckler [39] have consider two optimizations of the basic closure conversion strategy, called *selective* and *lightweight* closure conversion, and provide a correctness proof for each of these in an untyped setting. Hannan [9] recasts Wand's work into a typed setting and provides correctness proofs for one of Wand's optimizations. Hannan's translation, like ours, is given as a deductive system, but he does not consider the important issue of environment representation (preferring an abstract account instead), nor does he consider the typing properties of the closure-converted code. Finally, neither Wand nor Hannan consider closure conversion under a type-passing interpretation of polymorphism.

The remainder of this paper is organized as follows. In Section 2, we give an overview of closure conversion and the typing issues involved for the simply-typed λ-calculus. In Section 3, we provide the details of our type-preserving transform for the simply-typed case. In Section 4, we give an overview of closure conversion and the typing issues involved for the predicative fragment of the polymorphic λ-calculus. The formal development of this conversion is given in Section 5.

2 Overview of Simply-Typed Closure Conversion

The main ideas of closure conversion are illustrated by considering the following ML program:

```
let val x = 1
    val y = 2
    val z = 3
    val f = λw. x + y + w
in
    f 100
end.
```

The function `f` contains free variables `x` and `y`. We may eliminate references to these variables from the body of `f` by abstracting an environment `env` and replacing `x` and `y` by references to the environment. In compensation, a suitable environment containing the bindings for `x` and `y` must be passed to `f` before it is applied. This leads to the following translation:

```
let val x = 1
    val y = 2
    val z = 3
    val f = (λenv. λw. (#x env) + (#y env) + w)
            {x=x, y=y}
in
    f 100
end.
```

References to x and y in the body of f are replaced by projections (field selections) #x and #y that access the corresponding component of the environment. Since the code for f is closed, it may be hoisted out of the enclosing definition and defined at the top-level. We ignore this "hoisting" phase and instead concentrate on the process of closure conversion.

In the preceding example the environment contains bindings only for x and y, and is thus as small as possible. Since z is in scope, it is also sensible to include z in the environment of f, resulting in the following code:

```
let val x = 1
    val y = 2
    val z = 3
    val f = (λenv. λw. (#x env) + (#y env) + w)
            {x=x, y=y, z=z}
in
    f 100
end.
```

In the examples above, we used a flat FAM-like [4] representation of the environment as a record with one field for each variable. Alternatively we could choose a linked CAM-like [5] representation in which each binding is a separate frame attached to the front of the remaining bindings. This idea leads to the following translation:

```
let val x = 1
    val y = 2
    val z = 3
    val f = (λenv. λw.
               (#x(#link(#link env))) +
                (#y(#link env)) + w)
            {z=z, link={y=y, link={x=x}}}
in
    f 100
end.
```

The linked representation facilitates environment sharing, but accessing a variable requires link traversals proportional to the nesting depth of the variable in the environment. The linked representation also supports constant-time closure creation, but this requires reusing the current environment. Reusing the current environment can result in unnecessary bindings in the environment (such as z above), leading to space leaks.

These simple translations fail to delay the application of the code to its environment under call-by-value evaluation. A natural representation of a delayed application or closure is an ordered pair (code, env) consisting of the code together with its environment. Application of a closure to an argument proceeds by projecting the code part from the closure and then applying it simultaneously to both the environment and the argument according to some calling convention. For example,

```
let val x = 1
    val y = 2
    val z = 3
    val code = λenv. λw. #x(env) + #y(env) + w
    val env = {x=x, y=y}
    val f = (code, env)
in
    (#1 f) (#2 f) 100
end.
```

But since code has a type of the form $\tau_{\mathrm{ve}} \to \tau_1 \to \tau_2$, where τ_{ve} is the type of the environment env, the closure as a whole would have type $(\tau_{\mathrm{ve}} \to \tau_1 \to \tau_2) \times \tau_{\mathrm{ve}}$, exposing the type of the environment. As a result, this translation does not, in general, preserve types. For example, consider the following ML source program with type int $\to$ int:

```
let val y = 1
in
    if true then
        λx. x+y
    else
        λz. z
end.
```

Closure converting this expression and representing the closures as pairs yields

```
let val y = 1
in
    if true then
       (λenv. λx. x + #y(env), {y=y})
    else
       (λenv. λz. z, {})
end.
```

This program fails to type-check because the then-clause of the conditional has type $(\{\mathtt{y{:}int}\} \to \mathtt{int} \to \mathtt{int}) \times \{\mathtt{y{:}int}\}$, whereas the else-clause has type $(\{\} \to \mathtt{int} \to \mathtt{int}) \times \{\}$.

If types are to be preserved by closure conversion, the representation of the environment must be hidden. This may be achieved through the use of existential types [22], whose typing rules are given in Figure 1. Briefly, the pack construct packages a type τ with a term e, abstracting certain occurrences of τ in the type of e as the type variable t. The open operation extracts the contents of a package for use within a fixed scope, holding the type component of the package abstract. (See Mitchell and Plotkin's article [22] for further discussion of existential types.)

Using existentials, we may hide the type of the environment by abstracting it from the type of the closure itself. Specifically, a closure of type $\tau_1 \to \tau_2$ is represented by a package of the form

$$\mathtt{pack}\ \tau_{\mathrm{ve}}\ \mathtt{with}\ (\mathtt{code},\mathtt{env})\ \mathtt{as}\ \exists t_{\mathrm{ve}}.(t_{\mathrm{ve}} \to \tau_1 \to \tau_2) \times t_{\mathrm{ve}}$$

with type $\exists t_{\mathrm{ve}}.(t_{\mathrm{ve}} \to \tau_1 \to \tau_2) \times t_{\mathrm{ve}}$. Applying this to the example of the conditional expression given above, we obtain the translation

```
let val y = 1
in
  if true then
    pack {y:int} with (λenv. λx. x+#y(env),{y=y})
    as ∃t_ve.(t_ve → int → int) × t_ve
  else
    pack {} with (λenv. λz.z, {})
    as ∃t_ve.(t_ve → int → int) × t_ve
end.
```

It is easy to see that the types of the clauses of the conditional agree, and that the translation has type $\exists t_{\mathrm{ve}}.(t_{\mathrm{ve}} \to \mathtt{int} \to \mathtt{int}) \times t_{\mathrm{ve}}$.

With closures represented as packages of existential type, applications of the form e e' are translated as follows:

$$\frac{\Delta;\Gamma \vdash e : \sigma[\tau/t]}{\Delta;\Gamma \vdash \texttt{pack}\ \tau\ \texttt{with}\ e\ \texttt{as}\ \exists t.\sigma : \exists t.\sigma} \qquad \frac{\Delta;\Gamma \vdash e_1 : \exists t.\sigma' \quad \Delta \uplus \{t\};\Gamma \uplus \{x{:}\sigma'\} \vdash e_2 : \sigma}{\Delta;\Gamma \vdash \texttt{open}\ e_1\ \texttt{as}\ t\ \texttt{with}\ x\ \texttt{in}\ e_2 : \sigma}\,(t \notin FTV(\sigma), t \notin \Delta)$$

Figure 1: Typing Rules for Existentials

```
open e as t_ve with z : (t_ve → τ1 → τ2) × t_ve
in
    (#1 z) (#2 z) e'
end.
```

That is, the package e is opened, holding the environment representation abstract, and the code part is simultaneously applied to both the environment and the argument of the application.

3 A Formal Account of Simply-Typed Closure Conversion

In this section we present the details of closure conversion for the call-by-value, simply-typed λ-calculus. We break the full transformation into two stages, as outlined in the introduction. To simplify the presentation, we begin with a version of abstract closure conversion that does not admit sharing of environments and then consider the general, shared environment case separately. Next, we give the representation of closures in terms of existential types as sketched in the preceding section. Finally, we prove the correctness of the translations using a logical relations argument.

We define the syntax of the source language, $\lambda^{\rightarrow}$, as follows:

Types	$\tau ::= b \mid \tau_1 \rightarrow \tau_2$
Expressions	$e ::= c \mid x \mid \lambda x{:}\tau.\, e \mid e_1\, e_2$
Values	$v ::= c \mid \lambda x{:}\tau.\, e$

Types consist of base types (b) and function types[1]. Expressions consist of constants (c) of base type, variables, abstractions, and applications. We use Γ to denote a *sequence* of type bindings of the form $\{x_1{:}\tau_1, \ldots, x_n{:}\tau_n\}$ $(n \geq 0)$ where the x_i's are distinct variables. The judgement $\Gamma \vdash e : \tau$ asserts that the expression e has type τ under the type assignment Γ, and is derived from the standard typing rules of the simply-typed λ-calculus. The dynamic semantics of the language is defined by judgements of the form $e \hookrightarrow v$ asserting that the closed expression e evaluates to the value v. The judgement is defined by the following standard inference rules for call-by-value evaluation:

$$v \hookrightarrow v \qquad \frac{e_1 \hookrightarrow \lambda x{:}\tau_1.\, e \quad e_2 \hookrightarrow v_2 \quad e[v_2/x] \hookrightarrow v}{e_1\, e_2 \hookrightarrow v}$$

3.1 Abstract Closure Conversion

We define the target language for abstract closure conversion, λ^{cl}, as follows:

Types	$\tau ::=$	$b \mid \tau_1 \rightarrow \tau_2 \mid \langle \tau_1 \times \ldots \times \tau_n \rangle \mid \texttt{code}(\tau_{\rm ve}, \tau_1, \tau_2)$
Exp's	$e ::=$	$c \mid x \mid e_1 e_2 \mid \langle e_1, \ldots, e_n \rangle \mid \pi_i(e) \mid$
		$\lambda x_{\rm ve}{:}\tau_{\rm ve}.\, \lambda x{:}\tau_1.\, e \mid \langle\!\langle e_1, e_2 \rangle\!\rangle$
Values	$v ::=$	$c \mid \lambda x_{\rm ve}{:}\tau_{\rm ve}.\, \lambda x{:}\tau_1.\, e \mid \langle v_1, \ldots, v_n \rangle \mid \langle\!\langle v_1, v_2 \rangle\!\rangle$

[1]The results of this paper easily extend to other source types including products and sums.

In the introduction we informally presented a closure as a partial application of code to an environment, with the intention that this application is delayed until the closure is applied to an argument. To make this precise we introduce an explicit closure form, written $\langle\!\langle e, e_{\rm ve} \rangle\!\rangle$, where e is the code of the closure and $e_{\rm ve}$ is its environment. Notice that closures are distinguished from applications of functions to arguments, which are written in the usual way by juxtaposition. To capture the restriction that the code part of a closure should be closed, we introduce a special code type, $\texttt{code}(\tau_{\rm ve}, \tau_1, \tau_2)$, consisting of closed terms of the form $\lambda x_{\rm ve}{:}\tau_{\rm ve}.\lambda x{:}\tau_1.e$, which abstract both an environment and an argument[2].

The typing rules for λ^{cl} are standard except for code and closures, whose rules are given as follows:

$$\frac{\{x_{\rm ve}{:}\tau_{\rm ve}, x{:}\tau_1\} \vdash e : \tau_2}{\Gamma \vdash \lambda x_{\rm ve}{:}\tau_{\rm ve}.\lambda x{:}\tau_1.e : \texttt{code}(\tau_{\rm ve}, \tau_1, \tau_2)}$$

$$\frac{\Gamma \vdash e : \texttt{code}(\tau_{\rm ve}, \tau_1, \tau_2) \qquad \Gamma \vdash e_{\rm ve} : \tau_{\rm ve}}{\Gamma \vdash \langle\!\langle e, e_{\rm ve} \rangle\!\rangle : \tau_1 \rightarrow \tau_2}$$

The evaluation rules governing closures are given as follows:

$$\frac{e_1 \hookrightarrow v_1 \qquad e_2 \hookrightarrow v_2}{\langle\!\langle e_1, e_2 \rangle\!\rangle \hookrightarrow \langle\!\langle v_1, v_2 \rangle\!\rangle}$$

$$\frac{e_1 \hookrightarrow \langle\!\langle \lambda x_{\rm ve}{:}\tau_{\rm ve}.\lambda x{:}\tau_1.e, v_{\rm ve} \rangle\!\rangle \qquad e_2 \hookrightarrow v_2 \quad e[v_{\rm ve}/x_{\rm ve}, v_2/x] \hookrightarrow v}{e_1\ e_2 \hookrightarrow v}$$

When a closure is applied to an argument, the environment and the argument are substituted for the corresponding variables and the body of the code is evaluated.

We define abstract closure conversion as the type-directed translation from $\lambda^{\rightarrow}$ to λ^{cl} given in Figure 2. We formulate the translation as a deductive system with judgements of the form $\Gamma; x{:}\tau \vartriangleright e \rightsquigarrow e'$ and $\Gamma; x{:}\tau \vartriangleright \Gamma' \rightsquigarrow e'_{\rm ve}$, where Γ and Γ' are source type assignments, τ is a source type, e is a source expression, and e' and $e'_{\rm ve}$ are target expressions. The distinguished variable x is used to represent the argument of the nearest enclosing λ-abstraction; the variables in Γ include this λ-abstraction's free variables.

The judgement $\Gamma; x{:}\tau \vartriangleright e \rightsquigarrow e'$ asserts that e' is the translation of e under the assumption that $\Gamma \uplus \{x{:}\tau\} \vdash e : \tau'$ for some τ'. The judgement $\Gamma; x{:}\tau \vartriangleright \Gamma' \rightsquigarrow e'_{\rm ve}$ asserts that $e'_{\rm ve}$ is an expression that evaluates to the environment corresponding to Γ', under the assumption that each binding in Γ' occurs in $\Gamma \uplus \{x{:}\tau\}$. The order of bindings in Γ is important, because this determines the translation of both environments and free variables.

[2]In practice, a multi-argument λ-abstraction is used for code in the target language. However, the polymorphic case requires a more complicated construct that abstracts both values (λ) and types (Λ). For uniformity we use a curried presentation to abstract multiple arguments.

$$(const)\ \Gamma; x{:}\tau \triangleright c \leadsto c \qquad (arg)\ \Gamma; x{:}\tau \triangleright x \leadsto x \qquad (env)\ \{x_1{:}\tau_1, \ldots, x_n{:}\tau_n\}; x{:}\tau \triangleright x_i \leadsto \pi_i(x_{\mathrm{ve}})$$

$$(abs)\ \frac{\Gamma; x'{:}\tau' \triangleright \Gamma' \leadsto e_{\mathrm{ve}} \quad \Gamma'; x{:}\tau \triangleright e \leadsto e'}{\Gamma; x'{:}\tau' \triangleright \lambda x{:}\tau.e \leadsto \langle\!\langle \lambda x_{\mathrm{ve}}{:}|\Gamma'|.\ \lambda x{:}\tau.\ e', e_{\mathrm{ve}} \rangle\!\rangle} \qquad (app)\ \frac{\Gamma; x{:}\tau \triangleright e_1 \leadsto e_1' \quad \Gamma; x{:}\tau \triangleright e_2 \leadsto e_2'}{\Gamma; x{:}\tau \triangleright e_1\ e_2 \leadsto e_1'\ e_2'}$$

$$(context)\ \frac{\Gamma; x{:}\tau \triangleright x_1 \leadsto e_1 \quad \ldots \quad \Gamma; x{:}\tau \triangleright x_n \leadsto e_n}{\Gamma; x{:}\tau \triangleright \{x_1{:}\tau_1, \ldots, x_n{:}\tau_n\} \leadsto \langle e_1, \ldots, e_n \rangle} \quad (\Gamma \uplus \{x{:}\tau\} \vdash x_i : \tau_i)$$

Figure 2: Simply-Typed Abstract Closure Conversion

We use the variable x_{ve} to hold the environment argument of the current code body. Thus, we translate free variables to projections of x_{ve}. More precisely, according to rule (*env*), we translate a reference to the free variable x_i found in the ith position of the type assignment Γ to the ith projection of the variable x_{ve}. On the other hand, according to rule (*arg*), we translate a reference to the argument of the current code body to the distinguished argument variable x.

Under the assumptions $\Gamma; x'{:}\tau'$, we translate an abstraction $\lambda x{:}\tau.e$ to a closure according to the (*abs*) rule. To construct the environment of the closure, we choose a type assignment Γ' such that $\Gamma; x'{:}\tau' \triangleright \Gamma' \leadsto e_{\mathrm{ve}}$ is derivable via the (*context*) rule and $\Gamma'; x{:}\tau \triangleright e \leadsto e'$. In effect, these rules require that every binding in the closure's environment must be in scope (*i.e.*, in $\Gamma \uplus \{x'{:}\tau'\}$) and the environment is required to contain bindings for all of the free variables in the original function $\lambda x{:}\tau.\,e$. However, Γ' may also contain bindings for variables that are in scope but do not occur free in the function. Consequently, there are many choices for Γ', with the exact choice being influenced by time and space considerations.

We construct the environment of a closure via the (*context*) rule by translating each of the variables occurring in Γ' (namely $x_1, \cdots, x_n$) to the target expressions $e_1, \cdots, e_n$. We place the resulting expressions in a tuple $\langle e_1, \ldots, e_n \rangle$, to form the environment data structure of the closure. This representation of the environment has type $\langle \tau_1 \times \cdots \times \tau_n \rangle$, which we summarize by writing $|\Gamma'|$.

To produce the code of the closure, we translate the body of the source function under the strengthened assumptions $\Gamma'; x{:}\tau$, producing the body of the code, e'. We then abstract the environment and argument, yielding the translation $\lambda x_{\mathrm{ve}}{:}|\Gamma'|.\ \lambda x{:}\tau.\ e'$.

Using a dummy "current argument" to translate an entire closed program, it is easy to prove by induction on the derivation of the translation that the translation preserves the type of a program.

Theorem 1 *If* $\emptyset \vdash e{:}\tau$ *and* $\emptyset; x{:}b \triangleright e \leadsto e'$, *then* $\emptyset \vdash e' : \tau$.

To prove the correctness of the translation, we use a type-indexed family of logical relations relating closed source expressions to closed target expressions ($\sim$) and closed source values to closed target values ($\approx$). The relations are defined by induction on source types as follows:

$$\begin{array}{lll} e \sim_\tau e' & \text{iff} & e \hookrightarrow v \text{ and } e' \hookrightarrow v' \text{ and } v \approx_\tau v' \\ c \approx_b c & & \\ v \approx_{\tau_1 \to \tau_2} v' & \text{iff} & \text{for all } v_1 \approx_{\tau_1} v_1',\ v\ v_1 \sim_{\tau_2} v'\ v_1'. \end{array}$$

We extend the relation to finite source (γ) and target substitutions (γ'), mapping variables to their respective class of values. These relations are defined as follows:

$$\begin{array}{l} \gamma \approx_{\{x_1:\tau_1,\ldots,x_n:\tau_n\};x:\tau} [\langle v_1, \ldots, v_n \rangle / x_{\mathrm{ve}}, v/x] \\ \quad \text{iff } \gamma(x_i) \approx_{\tau_i} v_i \text{ for } 1 \le i \le n \text{ and } \gamma(x) \approx_\tau v. \end{array}$$

Theorem 2 *Let* $\gamma \approx_{\Gamma; x':\tau'} \gamma'$. *If* $\Gamma \uplus \{x'{:}\tau'\} \vdash e : \tau$ *and* $\Gamma; x'{:}\tau' \triangleright e \leadsto e'$, *then* $\gamma(e) \sim_\tau \gamma'(e')$.

Thus, for a closed program of base type, evaluating the program and its translation yields syntactically equivalent values.

3.2 Sharing Environments

Some implementations of functional programming languages share portions of an environment among closures in an effort to decrease space and closure creation time. In this section we extend the treatment of abstract closure conversion to allow for shared environments. We achieve this by imposing additional structure on environments to allow for nested representations.

The type assignments in the previous section (Γ) consist of a *flat* sequence of variable declarations. To provide for shared environment representations, we enrich the structure of type assignments to support *nested* type assignments as follows:

$$\Theta ::= \{x{:}\tau\} \mid \langle \Theta_1, \ldots, \Theta_m \rangle$$

A nested type assignment is either a single type binding or a sequence of nested type assignments. The environment corresponding to the type assignment Θ has target language type $|\Theta|$, where $|\{x{:}\tau\}| = \tau$ and $|\langle \Theta_1, \ldots, \Theta_m \rangle| = \langle |\Theta_1| \times \ldots \times |\Theta_m| \rangle$. We can obtain a non-nested type assignment (Γ) from a nested type assignment (Θ) simply by dropping the extra structure.

We give the most important translation rules for closure conversion with nested environments in Figure 3; the remaining rules may be obtained from those in Figure 2 by replacing Γ with Θ throughout.

We use the (*env-tuple*) rule to construct a nested environment $\langle e_1, \cdots, e_n \rangle$ corresponding to the type assignments $\Theta_1, \cdots, \Theta_n$, if $\Theta; x{:}\tau \triangleright \Theta_i \leadsto e_i$, for $1 \le i \le n$. We obtain each of the Θ_i and e_i from the (*arg*), (*env*), (*subenv*), and (*env-tuple*) rules. We use the (*arg*) rule to translate the argument of the nearest enclosing λ-abstraction as an environment, and we use the (*env*) rule to translate the free variables of this abstraction as an environment. As before, we use the distinguished variables x and x_{ve} to hold these two values in the translation. We use the (*subenv*) rule to translate access to a type assignment nested within Θ to a

$$(arg)\quad \{x':\tau'\}; x{:}\tau \triangleright \{x{:}\tau\} \rightsquigarrow x \qquad (env)\quad \Theta; x{:}\tau \triangleright \Theta \rightsquigarrow x_{\mathrm{ve}} \qquad (var)\quad \frac{\Theta; x{:}\tau \triangleright \{x':\tau'\} \rightsquigarrow e}{\Theta; x{:}\tau \triangleright x' \rightsquigarrow e}$$

$$(subenv)\quad \frac{\Theta_i; x{:}\tau \triangleright \Theta \rightsquigarrow e}{\langle \Theta_1, \ldots, \Theta_n \rangle; x{:}\tau \triangleright \Theta \rightsquigarrow e[\pi_i(x_{\mathrm{ve}})/x_{\mathrm{ve}}]} \quad (env\text{-}tuple)\quad \frac{\Theta; x{:}\tau \triangleright \Theta_1 \rightsquigarrow e_1 \quad \cdots \quad \Theta; x{:}\tau \triangleright \Theta_n \rightsquigarrow e_n}{\Theta; x{:}\tau \triangleright \langle \Theta_1, \ldots, \Theta_n \rangle \rightsquigarrow \langle e_1, \ldots, e_n \rangle}$$

Figure 3: Simply-Typed Closure Conversion using Nested Environments

projection of the environment corresponding to Θ. Finally, we translate access to a variable within a type assignment via the (var) rule.

As an example, consider the translation

$$\begin{array}{l}\langle \{x_1{:}int\}, \{x_2{:}int\} \rangle; x'{:}int \triangleright (\lambda x{:}int.x' + x_1 + x_2) \rightsquigarrow \\ \quad \langle\!\langle \lambda x_{\mathrm{ve}}{:}\tau.\lambda x{:}int.\pi_1(x_{\mathrm{ve}}) + \pi_1(\pi_2(x_{\mathrm{ve}})) + \pi_2(\pi_2(x_{\mathrm{ve}})), \\ \qquad \langle x', x_{\mathrm{ve}} \rangle \rangle\!\rangle \end{array}$$

where τ is $\langle int \times \langle int \times int \rangle \rangle$. We construct the new environment for the closure by pairing the current argument x' and the current environment x_{ve} according to the $(env\text{-}tuple)$ rule. If we used the flat translation given in Figure 2, then we would have to project the values for x_1 and x_2 out of the current environment and place these values and the current argument into a newly allocated tuple.

Nested type assignments are sufficiently flexible to handle many commonly-used environment representations. For example, the Categorical Abstract Machine, or CAM [5], uses a linked list to represent the environment. This is reflected in our framework by restricting the shape of nested type assignments and by restricting the $(env\text{-}tuple)$ rule to "cons" the current argument onto the current environment, as follows:

$$(CAM\ context)\qquad \Theta_c ::= \{x{:}\tau\} \mid \langle \{x{:}\tau\}, \Theta_c \rangle$$

$$(env\text{-}tuple)\qquad \Theta_c; x{:}\tau \triangleright \langle x{:}\tau, \Theta_c \rangle \rightsquigarrow \langle x, x_{\mathrm{ve}} \rangle.$$

The advantage of the CAM strategy is that the cost of the construction of a new environment is constant. However, in the worst case, accessing values in the environment takes time proportional to the length of the environment.

In contrast, the FAM [4] uses flat environments with no sharing. The closure conversion of Figure 2 accurately models the environment strategy of the FAM if we choose a specific strengthening strategy in the (abs) rule where only the free variables of the function are preserved in the resulting closure's environment. The advantage of the FAM environment representation is that the cost of variable lookup is always constant and the representation is "safe for space" [1] according to Appel's definition. However, constructing the environment for a closure takes time proportional to the number of free variables in the function, and closures cannot share portions of their environment.

Clearly, there are a variety of other strategies for forming environments. For example, the shared closure strategy described by Appel and Shao [30] that is also safe for space can also be formulated in our framework. However, to determine a good representation for each closure's environment requires a good deal more information including an estimate as to how many times each variable is accessed, when garbage collection can occur, what garbage collection algorithm is used, *etc.*

3.3 Closure Representation

The purpose of abstract closure conversion is to choose an environment representation for each closure and to make the construction of closures explicit. By making environments explicit, we expose operations that are implicit at the source level to an optimizer at the target level. In particular, an optimizer can eliminate redundant constructions of environments or redundant projections from environments.

However, the process of extracting the code and environment of a closure remains an implicit, atomic operation of the operational semantics. Hence, we cannot optimize these closure operations. For instance, if the same closure is repeatedly applied in a loop, it is not possible to extract the code and environment once, repeating only the application to the environment and argument within the loop.

To make such optimizations possible, we choose a representation of closures in terms of generic primitives that would, in practice, already be present in the intermediate language. Specifically, we consider a target language $\lambda^{\exists}$ with existential types, defined by the following grammar:

$$\begin{array}{lrl} Types & \tau ::= & b \mid t \mid \langle \tau_1 \times \ldots \times \tau_n \rangle \mid \\ & & \mathtt{code}(\tau_{\mathrm{ve}}, \tau_1, \tau_2) \mid \exists t.\tau \\ \\ Exp's & e ::= & c \mid e_1(e_2, e_3) \mid \lambda x_{\mathrm{ve}}{:}\tau_{\mathrm{ve}}.\lambda x{:}\tau_1.e \mid \\ & & \langle e_1, \ldots, e_n \rangle \mid \pi_i\ e \mid \\ & & \mathtt{pack}\ \tau\ \mathtt{with}\ e\ \mathtt{as}\ \tau \mid \\ & & \mathtt{open}\ e_1\ \mathtt{as}\ t\ \mathtt{with}\ x{:}\tau\ \mathtt{in}\ e_2 \end{array}$$

This language includes existential types and code types, but not function types; we show how to define function types in terms of these primitive constructs. We restrict applications to the form $e_1(e_2, e_3)$ in order to preclude a partial application of code to its environment; this can be seen as a specialized use of multi-argument functions.

Typing judgements for $\lambda^{\exists}$ are of the form $\Delta; \Gamma \vdash e : \tau$ where Δ is a list of type variables in scope and Γ is a type assignment for variables in scope. We assume that the free type variables of the types in the range of Γ and the free type variables of e and τ are contained in Δ. The typing rules and evaluation rules of the language are standard (see [22] and Figure 1).

We describe the closure representation phase in two parts. We begin by defining a translation from λ^{cl} to $\lambda^{\exists}$ types, denoted $|\tau|$, as follows:

$$\begin{array}{rcl} |b| & = & b \\ |\langle \tau_1 \times \ldots \times \tau_n \rangle| & = & \langle |\tau_1| \times \ldots \times |\tau_n| \rangle \\ |\mathtt{code}(\tau_{\mathrm{ve}}, \tau_1, \tau_2)| & = & \mathtt{code}(|\tau_{\mathrm{ve}}|, |\tau_1|, |\tau_2|) \\ |\tau_1 \rightarrow \tau_2| & = & \exists t_{\mathrm{ve}}.\langle \mathtt{code}(t_{\mathrm{ve}}, |\tau_1|, |\tau_2|) \times t_{\mathrm{ve}} \rangle. \end{array}$$

We translate an arrow type to a pair consisting of code and an environment, with the environment type held abstract using an existential quantifier.

$$(closure) \quad \frac{\Gamma \rhd e : \mathtt{code}(\tau_{\mathrm{ve}}, \tau_1, \tau_2) \rightsquigarrow e' \qquad \Gamma \rhd e_{\mathrm{ve}} : \tau_{\mathrm{ve}} \rightsquigarrow e'_{\mathrm{ve}}}{\Gamma \rhd \langle\!\langle e, e_{\mathrm{ve}} \rangle\!\rangle : \tau_1 \to \tau_2 \rightsquigarrow \mathtt{pack}\ |\tau_{\mathrm{ve}}|\ \mathtt{with}\ \langle e', e'_{\mathrm{ve}} \rangle\ \mathtt{as}\ |\tau_1 \to \tau_2|}$$

$$(app) \quad \frac{\Gamma \rhd e_1 : \tau_1 \to \tau_2 \rightsquigarrow e'_1 \qquad \Gamma \rhd e_2 : \tau_1 \rightsquigarrow e'_2}{\begin{array}{c}\Gamma \rhd e_1 e_2 : \tau_2 \rightsquigarrow \\ \mathtt{open}\ e'_1\ \mathtt{as}\ t_{\mathrm{ve}}\ \mathtt{with}\ x{:}\langle \mathtt{code}(t_{\mathrm{ve}}, |\tau_1|, |\tau_2|) \times t_{\mathrm{ve}} \rangle\ \mathtt{in}\ (\pi_1\ x)(\pi_2\ x, e'_2)\end{array}} \ (x \notin Dom(\Gamma))$$

Figure 4: Important Rules of Simply-Typed Closure Representation

Next, we define the translation of λ^{cl} terms to $\lambda^{\exists}$ terms in Figure 4. The judgements of the translation are of the form $\Gamma \rhd e : \tau \rightsquigarrow e'$, where Γ, e, and τ are a λ^{cl} type assignment, expression, and type respectively, and e' is a $\lambda^{\exists}$ expression. The interesting rules are (*closure*) and (*app*). The other rules (not shown) simply map the other λ^{cl} constructs to their $\lambda^{\exists}$ counterparts. We translate a closure to a pair of the code and the environment packed with the type of the environment. We translate an application to an `open`, extract from a package the pair of a code and an environment, and then apply the code to the environment and the argument.

It is easy prove that the translation preserves the type of a program up to the translation of the type. We do so by first extending the type translation to type assignments, setting

$$|\{x_1{:}\tau_1, \ldots, x_n{:}\tau_n\}| = \{x_1{:}|\tau_1|, \ldots, x_n{:}|\tau_n|\}.$$

Theorem 3 *If* $\Gamma \vdash e : \tau$ *and* $\Gamma \rhd e : \tau \rightsquigarrow e'$, *then* $\emptyset; |\Gamma| \vdash e' : |\tau|$.

Correctness of the translation is proven using logical relations between λ^{cl} and $\lambda^{\exists}$ expressions, λ^{cl} and $\lambda^{\exists}$ values, and λ^{cl} and $\lambda^{\exists}$ substitutions. The definition of the relations and the proof of the correctness can be found in our technical report [21].

4 Overview of Polymorphic Closure Conversion

Closure conversion for a language with ML-style (*i.e.*, predicative [13]) explicit polymorphism follows a similar pattern to the simply-typed case, but with two additional complications. First, we must account for free type variables as well as free value variables in the code of an abstraction. Second, we must create closures for both value abstractions (λ-terms) and type abstractions (Λ-terms). In this section, we give an overview of the typing difficulties encountered when closure converting value abstractions; the treatment of type abstractions is similar (see Section 5 for details).

To eliminate free occurrences of type variables and ordinary variables from the code, we abstract with respect to a type environment and a value environment, replacing free variables by references to the appropriate environment. This process results in closed code that can be hoisted to the top level and shared among multiple closures. The code is partially applied to suitable representations of the type and value environments to form a polymorphic closure. As in the simply-typed case, we need a data structure to represent the delayed partial application of the code to its type and value environments. In addition, we must abstract both the kind of the type environment and the type of the value environment so that their representations remain private to the closure. Without the abstraction, we run into the same typing problems that we encountered in the simply-typed case.

As a running example, consider the expression

```
λx:t₁. (x:t₁, y:t₂, z:int),
```

where t_1 and t_2 are free type variables and `y` and `z` are free value variables of type t_2 and `int` respectively. It is easy to check that this expression has type $t_1 \to (t_1 \times t_2 \times \mathtt{int})$. To closure convert the expression, we translate it to the partial application

```
let val code =
      Λtenv :: {t₁::Ω, t₂::Ω}.
        λvenv : {y:#t₂ tenv, z:int}.
          λx : (#t₁ tenv).(x, #y venv, #z venv)
in
    code {t₁=t₁, t₂=t₂} {y=y, z=z}
end.
```

The code of the closure abstracts a type environment `tenv` and a value environment `venv`. The actual type environment $\{t_1{=}t_1, t_2{=}t_2\}$ is a record of kind $\{t_1{::}\Omega, t_2{::}\Omega\}$, where Ω is the kind of monotypes. The actual value environment `{y=y, z=z}` is a record with type $\{\mathtt{y}{:}t_2,\ \mathtt{z{:}int}\}$. Note, however, that this type contains a free reference to t_2, which must be replaced by a reference to the type environment in order to ensure that the translated code is closed. We therefore ascribe the type $\{\mathtt{y{:}\#}t_2\ \mathtt{tenv},\ \mathtt{z{:}int}\}$ to the value environment, noting that the projection $\mathtt{\#}t_2\ \mathtt{tenv}$ is equivalent to t_2 when the actual type environment is as given earlier. By similar reasoning we assign the type $\mathtt{\#}t_1\ \mathtt{tenv}$ to the argument `x` of the λ-abstraction. It is easy to check that the code of the closure has the type σ_{code} given by the equation

```
σ_code =
  ∀tenv::{t₁::Ω, t₂::Ω}.
    {y:#t₂ tenv, z:int}→
      (#t₁ tenv)→((#t₁ tenv)×(#t₂ tenv)×int).
```

It follows that the entire `let` expression has the type of the original term, namely $t_1 \to (t_1 \times t_2 \times \mathtt{int})$.

Now let us consider the representation of the partial application of `code` to its type and value environments as a data structure. This data structure must be "mixed phase" in the sense that it consists of both type and value components. This suggests using a package of existential type of the form

$$e = \texttt{pack}\ \{t_1\texttt{=}t_1,\ t_2\texttt{=}t_2\}\ \texttt{with (code, \{y=y, z=z\})}$$
$$\texttt{as}\ \exists t_{\rm te}\texttt{::}\kappa_{\rm te}\texttt{.}\sigma_{\rm code} \times \tau_{\rm ve}\texttt{,}$$

where `code` is as given earlier and

$$\kappa_{\rm te} = \{t_1\texttt{::}\Omega,\ t_2\texttt{::}\Omega\}$$
$$\tau_{\rm ve} = \{\texttt{y:\#}t_2\ t_{\rm te},\ \texttt{z:int}\}.$$

This package is well-typed according to the usual rules for existentials.

In contrast, consider what happens when we attempt to give the translation of the application of e to an argument e' of type t_1. Proceeding as in the simply-typed case, we introduce an `open` expression that extracts the code, the type environment, and the value environment from the closure, and applies the code to the environments and argument. Doing so results in the following translation:

$$\begin{array}{l}\texttt{open}\ e\ \texttt{as}\ t_{\rm te}\texttt{::}\kappa_{\rm te}\ \texttt{with w:}\sigma_{\rm code} \times \tau_{\rm ve}\\ \texttt{in}\\ \quad \texttt{(\#1 w)}\ t_{\rm te}\ \texttt{(\#2 w)}\ e'\\ \texttt{end.}\end{array}$$

Unfortunately, this expression is not well-typed. The difficulty is that e' has type t_1, whereas the expression `(#1 w)` $t_{\rm te}$ `(#2 w)` has type

$$(\texttt{\#}t_1\ t_{\rm te}) \to ((\texttt{\#}t_1\ t_{\rm te}) \times (\texttt{\#}t_2\ t_{\rm te}) \times \texttt{int}).$$

Since $t_{\rm te}$ is abstract, the type variable t_1 is not equivalent to $\texttt{\#}t_1\ t_{\rm te}$. Consequently, the proposed translation of application fails to type-check.

One way to get around this problem is to apply the code to the type environment before forming the closure. This yields

$$\begin{array}{l}\texttt{let val c = code}\ \{t_1\texttt{=}t_1,t_2\texttt{=}t_2\}\\ \texttt{in}\\ \quad \texttt{pack}\ \{t_1\texttt{=}t_1,t_2\texttt{=}t_2\}\ \texttt{with (c, \{y=y,z=z\})}\\ \quad \texttt{as}\ \exists t_{\rm te}\texttt{::}\kappa_{\rm te}\texttt{.}\sigma_{\rm c} \times \tau_{\rm ve}\\ \texttt{end,}\end{array}$$

where the type $\sigma_{\rm c}$ is given by the equation

$$\sigma_{\rm c} = \{\texttt{y:\#}t_1\ t_{\rm te}\texttt{,z:int}\} \to (\texttt{\#}t_1\ t_{\rm te}) \to ((\texttt{\#}t_1\ t_{\rm te}) \times (\texttt{\#}t_2\ t_{\rm te}) \times \texttt{int}).$$

The translation of application given above will work in this case because the code and the value environment both use $t_{\rm te}$ as the type environment. But this approach depends upon the very mechanism we are attempting to eliminate, namely *partial application*. The partial application of the code to the type environment produces code that is no longer closed. Thus, the code cannot be shared among the different instantiations of the type environment.

Our solution to this issue is to *constrain* the code so that it can be applied to a closure's value environment only when it is also applied to the *same* closure's type environment. This ensures that the type environment passed to the code and the type environment used in the construction of the closure's value environment are the same. Fortunately, typing constraints of this form have already been addressed by research on module systems [20, 19, 11].

Following Harper and Lillibridge [11], we use the notion of *translucent* types to express the desired constraint on the code. In particular, when forming the closure, we coerce the code to have the translucent type

$$\begin{array}{l}\forall\texttt{tenv} = \{t_1\texttt{=}t_1,t_2\texttt{=}t_2\}\texttt{::}\kappa_{\rm te}\texttt{.}\\ \quad \{\texttt{y:\#}t_1\ \texttt{tenv,z:int}\} \to (\texttt{\#}t_1\ \texttt{tenv}) \to\\ \qquad ((\texttt{\#}t_1\ \texttt{tenv}) \times (\texttt{\#}t_2\ \texttt{tenv}) \times \texttt{int}).\end{array}$$

This type is a super-type of the original code type $\sigma_{\rm code}$ because we have constrained the bound type variable `tenv` to be bound to a particular type, namely the type environment of the closure. (See Harper and Lillibridge [11] and Leroy [19] for further discussion of subtyping in this setting.) This constraint ensures that this reference to the code will only be applied to the type environment of the closure.

The constraint on `tenv` allows us to conclude that $\texttt{\#}t_1$ `tenv` is equivalent to t_1 and similarly, that $\texttt{\#}t_2$ `tenv` is equivalent to t_2. We propagate these equivalences into the type yielding

$$\begin{array}{l}\forall\texttt{tenv} = \{t_1\texttt{=}t_1,t_2\texttt{=}t_2\}\texttt{::}\kappa_{\rm te}\texttt{.}\\ \quad \{\texttt{y:}t_1\texttt{,z:int}\} \to t_1 \to (t_1 \times t_2 \times \texttt{int}).\end{array}$$

We can now form the package containing the type environment, code, and value

$$\begin{array}{l}\texttt{pack}\ \{t_1\texttt{=}t_1,t_2\texttt{=}t_2\}\ \texttt{with (code, env)}\\ \texttt{as}\ \exists t_{\rm te}\texttt{::}\kappa_{\rm te}\texttt{.}\sigma \times \tau_{\rm ve}\texttt{,}\end{array}$$

where σ is given via the equation

$$\sigma = \forall\texttt{tenv=}t_{\rm te}\texttt{::}\kappa_{\rm te}\texttt{.}\{\texttt{y:\#}t_1\ t_{\rm te}\texttt{,z:int}\} \to (\texttt{\#}t_1\ t_{\rm te}) \to ((\texttt{\#}t_1\ t_{\rm te}) \times (\texttt{\#}t_2\ t_{\rm te}) \times \texttt{int}),$$

and show that this package has type $\exists t_{\rm te}{::}\kappa_{\rm te}.\sigma \times \tau_{\rm ve}$. Note that σ is the same as $\sigma_{\rm c}$, (the type of the partial application of `code` to the type environment), except for the additional constrained type abstraction of `tenv`. Through the use of translucency, we have accomplished the effect of partial application at the type-level without actually performing the application at the term-level.

Opening a package e of type $\exists t_{\rm te}{::}\kappa_{\rm te}.\sigma \times \tau_{\rm ve}$ to apply to an argument e' of type t_1 yields:

$$\begin{array}{l}\texttt{open}\ e\ \texttt{as}\ t_{\rm te}\texttt{::}\kappa_{\rm te}\texttt{, w:}\sigma \times \tau_{\rm ve}\\ \texttt{in}\\ \quad \texttt{(\#1 w)}\ t_{\rm te}\ \texttt{(\#2 w)}\ e'\\ \texttt{end.}\end{array}$$

The expression `(#1 w)` $t_{\rm te}$ `(#2 w)` e' has type $t_1 \to (t_1 \times t_2 \times \texttt{int})$, thus the entire expression is well-formed.

In summary, we use transluceny to constrain the type of code before placing it in a closure. We use pack to represent the mixed-phase data structure containing the code, type environent, and the value environment. The resulting package has a type of the form

$$(\exists t_{\rm te}{::}\kappa_{\rm te}.\forall t{=}t_{\rm te}{::}\kappa_{\rm te}.\tau_{\rm ve} \to \tau_1 \to \tau_2) \times \tau_{\rm ve}.$$

To avoid the typing problems encountered in the simply-typed case, we need to hide the representations of the value environment and the type environment. Thus, we use `pack` again to abstract the kind of the type environment and the type of the value environment, resulting in the following type for closures:

$$\exists k_{\rm te}.\exists t_{\rm ve}{::}\Omega.\exists t_{\rm te}{::}k_{\rm te}.(\forall t{=}t_{\rm te}{::}k_{\rm te}.t_{\rm ve} \to \tau_1 \to \tau_2) \times t_{\rm ve}.$$

Careful consideration of the foregoing discussion reveals that we only made limited use of translucency. The universally quantified variable `tenv` does not occur in the scope of the abstraction once the equational constraint on `tenv` is propagated. We use this property to provide a substantially simpler mechanism than the full translucent type calculus. In particular, we only need to capture the restriction that a polymorphic function must be applied to a specific type argument. This may be expressed by introducing a type $\tau \Rightarrow \sigma$ consisting of functions that must be applied to the constructor τ to yield a value of type σ. The following two rules govern this new type constructor:

$$\frac{\Delta \vdash \tau :: \kappa \quad \Delta;\Gamma \vdash e : \forall t{::}\kappa.\sigma}{\Delta;\Gamma \vdash e : \tau \Rightarrow \sigma[\tau/t]} \qquad \frac{\Delta;\Gamma \vdash e : \tau \Rightarrow \sigma}{\Delta;\Gamma \vdash e\,\tau : \sigma}$$

The first rule restricts the domain of type application to the specific constructor τ. This corresponds to restricting the type to $\forall t = \tau.\sigma$ and propagating the equivalence $t = \tau$ into σ. The actual type application for $\tau \Rightarrow \sigma$ is permitted only for constructors equivalent to τ. These two rules naturally come from the necessity of delaying type applications for closure conversion. Using this notation, the type translation of $\tau_1 \to \tau_2$ becomes

$$\exists k_{\mathrm{te}}.\exists t_{\mathrm{ve}}{::}\Omega.\exists t_{\mathrm{te}}{::}k_{\mathrm{te}}.(t_{\mathrm{te}} \Rightarrow t_{\mathrm{ve}} \to \tau_1 \to \tau_2) \times t_{\mathrm{ve}}.$$

The type of closures abstracts the kind of the type environment and the type of the value environment, ensuring that these may be chosen separately for each closure in the system. As in the simply-typed case, we have obtained an "object oriented" representation of polymorphic closures by exploiting a combination of the type systems proposed by Pierce and Turner [26] for objects and by Harper and Lillibridge [11] for modules.

5 A Formal Account of Polymorphic Closure Conversion

In this section, we present closure conversion for the predicative fragment of the second order λ-calculus. This fragment is sufficient to model Standard ML [13], and admits relatively simple correctness proofs based on logical relations. Our results extend to the full impredicative polymorphic λ-calculus, but at the expense of a substantially more complex correctness argument (based on Girard's method of candidates [7]).

We define the syntax of the source language $\lambda^{\forall}$ as follows:

Kinds	$\kappa ::= \Omega$
Constructors	$\tau ::= b \mid t \mid \tau_1 \to \tau_2$
Types	$\sigma ::= \tau \mid \sigma_1 \to \sigma_2 \mid \forall t{::}\kappa.\sigma$
Expressions	$e ::= c \mid x \mid \lambda x{:}\sigma_1.e \mid \Lambda t{::}\kappa.e \mid e_1\ e_2 \mid e\ \tau$
Values	$v ::= c \mid \lambda x{:}\sigma_1.e \mid \Lambda t{::}\kappa.e$

We use kinds (κ) to describe constructors (τ) and types (σ) to describe expressions (e). There is only one kind (Ω) for $\lambda^{\forall,cl}$, but since subsequent languages have a richer kind structure, we introduce kinds here for uniformity. Closed constructors of kind Ω correspond to a subset of types, in particular the types that do not include quantifiers (the monotypes). Thus, constructors of kind Ω can be injected into types. We leave this injection implicit and treat τ as both a constructor and a type.

A kind assignment Δ is a sequence that maps type variables to kinds and is of the form $\{t_1{::}\kappa_1,\ldots,t_n{::}\kappa_n\}$, $(n \geq 0)$. Typing judgements are of the form $\Delta;\Gamma \vdash e : \sigma$ where the free type variables of Γ, e, and σ are contained in the domain of Δ, and the free value variables of e are contained in the domain of Γ. Typing judgements are derived according to the standard typing rules of the second-order λ-calculus (see for example [13, 14]). The most interesting rules are the introduction and elimination rules for quantified types:

$$\frac{\Delta \uplus \{t{::}\kappa\};\Gamma \vdash e : \sigma}{\Delta;\Gamma \vdash \Lambda t{::}\kappa.\,e : \forall t{::}\kappa.\,\sigma} \qquad (t \notin Dom(\Delta))$$

$$\frac{\Delta;\Gamma \vdash e : \forall t{::}\kappa.\,\sigma}{\Delta;\Gamma \vdash e\ \tau : \sigma[\tau/t]} \qquad (FTV(\tau) \subseteq Dom(\Delta))$$

5.1 Abstract Closure Conversion

Abstract closure conversion for $\lambda^{\forall}$ converts both λ-abstractions and Λ-abstractions into abstract closures consisting of code, a type environment and a value environment. We consider here only flat environment representations, but note that the treatment of nested environments given in Section 3 carries over to the polymorphic case.

We define the syntax of the target language $\lambda^{\forall,cl}$ as follows:

Kinds	$\kappa ::=$	$\Omega \mid \langle \kappa_1 \times \ldots \times \kappa_2 \rangle$
Con's	$\tau ::=$	$b \mid t \mid \tau_1 \to \tau_2 \mid \langle \tau_1 \times \ldots \times \tau_n \rangle \mid$
		$\langle \tau_1,\ldots,\tau_n \rangle \mid \pi_i\,\tau \mid$
Types	$\sigma ::=$	$\tau \mid \sigma_1 \to \sigma_2 \mid \forall t{::}\kappa.\sigma \mid \langle \sigma_1 \times \ldots \times \sigma_n \rangle \mid$
		$\mathtt{vcode}(t_{\mathrm{te}}{::}\kappa_{\mathrm{te}},\sigma_{\mathrm{ve}},\sigma_1,\sigma_2) \mid$
		$\mathtt{tcode}(t_{\mathrm{te}}{::}\kappa_{\mathrm{te}},\sigma_{\mathrm{ve}},t{::}\kappa,\sigma)$
Exp's	$e ::=$	$c \mid x \mid e_1\ e_2 \mid e\ \tau \mid \langle e_1,\ldots,e_n \rangle \mid \pi_i\,e \mid$
		$\Lambda t_{\mathrm{te}}{::}\kappa_{\mathrm{te}}.\lambda x_{\mathrm{ve}}{:}\sigma_{\mathrm{ve}}.\lambda x{:}\sigma_1.e \mid$
		$\Lambda t_{\mathrm{te}}{::}\kappa_{\mathrm{te}}.\lambda x_{\mathrm{ve}}{:}\sigma_{\mathrm{ve}}.\Lambda t{::}\kappa.e \mid \langle\!\langle e_1,\tau,e_2 \rangle\!\rangle$

We use product kinds of the form $\langle \kappa_1 \times \ldots \times \kappa_n \rangle$ to specify the shapes of type environments in much the same way that we use product types to specify the shapes of value environments.

There are two sorts of code: code corresponding to an ordinary λ-abstraction has the form $\Lambda t_{\mathrm{te}}{::}\kappa_{\mathrm{te}}.\lambda x_{\mathrm{ve}}{:}\sigma_{\mathrm{ve}}.\lambda x{:}\sigma_1.e$ while code corresponding to a type abstraction has the form $\Lambda t_{\mathrm{te}}{::}\kappa_{\mathrm{te}}.\lambda x_{\mathrm{ve}}{:}\sigma_{\mathrm{ve}}.\Lambda t{::}\kappa.e$. The code in each case abstracts a type environment and a value environment. For the λ-case the code also abstracts a value argument, and for the Λ-case the code abstracts a type argument. We introduce the types `vcode` and `tcode` to distinguish the two types of code from the types of closures, to ensure closure conditions on code, and to preclude partial applications of code to environments. These types may be described by the following informal correspondences:

$$\mathtt{vcode}(t_{\mathrm{te}}{::}\kappa_{\mathrm{te}},\sigma_{\mathrm{ve}},\sigma_1,\sigma_2) \approx \forall t_{\mathrm{te}}{::}\kappa_{\mathrm{te}}.\sigma_{\mathrm{ve}} \to \sigma_1 \to \sigma_2$$
$$\mathtt{tcode}(t_{\mathrm{te}}{::}\kappa_{\mathrm{te}},\sigma_{\mathrm{ve}},t{::}\kappa,\sigma) \approx \forall t_{\mathrm{te}}{::}\kappa_{\mathrm{te}}.\sigma_{\mathrm{ve}} \to \forall t{::}\kappa.\sigma.$$

We consider code types to be polymorphic, so these types do not lie in the range of a polymorphic quantifier.[3]

Abstract closures have the form $\langle\!\langle e_1,\tau,e_2 \rangle\!\rangle$, consisting of piece of code e_1, a type environment τ, and a value environment e_2.

For the typing of $\lambda^{\forall,cl}$, kind assignments (Δ) map type variables to kinds while type assignments (Γ) map value variables to types. The judgements of the static semantics are as follows:

$\Delta \vdash \tau :: \kappa$	τ is a well-formed constructor of kind κ.
$\Delta \vdash \sigma$	σ is a well-formed type.
$\Delta \vdash \tau_1 \equiv \tau_2 :: \kappa$	τ_1 and τ_2 are equivalent constructors.
$\Delta \vdash \sigma_1 \equiv \sigma_2$	σ_1 and σ_2 are equivalent types.
$\Delta;\Gamma \vdash e : \sigma$	e is a well-formed expression of type σ.

[3] This restriction is relaxed in the impredicative case.

The formation rules of types are standard. We have to introduce definitional equality of constructors and types to account for projections of constructors from product kinds. These rules consist of the equivalence rules for projections below, as well as the standard rules for equivalence and congruence:

$$\Delta \vdash \pi_i\langle\tau_1,\ldots,\tau_n\rangle \equiv \tau_i :: \kappa_i$$
$$\Delta \vdash \langle\pi_1\ \tau,\ldots,\pi_n\ \tau\rangle \equiv \tau :: \langle\kappa_1 \times \ldots \times \kappa_n\rangle$$

The typing rules for expressions are standard except for the rules for codes and closures. These rules are defined in Figure 5. We require that code values be closed with respect to both type variables as well as value variables. This allows us to share the code among multiple instantiations of the free type variables and free value variables.

We define abstract closure conversion from $\lambda^{\forall}$ to $\lambda^{\forall,cl}$ by the deductive system given in Figures 6 and 7. The judgement $\Delta_{\text{env}};\Delta_{\text{arg}} \rhd \sigma \rightsquigarrow \sigma'$ means that σ' is the translation of σ where Δ_{env} is a kind assignment corresponding to a type environment and Δ_{arg} is a kind assignment corresponding to a type argument (if any). This judgement also implicitly defines a translation from constructors to constructors, since source-level constructors (τ) are a subset of types (σ) and the translation maps constructors to constructors. In translated programs the type variable t_{te} is used for type environments.

The judgement $\Delta_{\text{env}};\Delta_{\text{arg}};\Gamma_{\text{env}};\Gamma_{\text{arg}} \rhd e \rightsquigarrow e'$ means e' is a translation of e where Δ_{env} and Δ_{arg} are as in the type translation, and Γ_{env} and Γ_{arg} are type assignments corresponding to the value environment and value argument respectively. A type environment corresponding to Δ_{env} and a value environment corresponding to Γ_{env} are implemented in the target language by types of the form $|\Delta_{\text{env}}|$ and $|\Gamma_{\text{env}}|$ respectively, as defined below:

$$\begin{array}{rcl} |\{t_1{::}\kappa_1,\ldots,t_n{::}\kappa_n\}| & = & \langle\kappa_1 \times \ldots \times \kappa_n\rangle \\ |\{x_1{:}\sigma_1,\ldots,x_n{:}\sigma_n\}| & = & \langle\sigma_1 \times \ldots \times \sigma_n\rangle. \end{array}$$

The most interesting rules are the term translations of value and type abstractions. In each case, an appropriate type environment and value environment must be constructed as part of the closure. Thus, assignments Δ'_{env} and Γ'_{env} must be chosen as subsets of the current assignments $\Delta_{\text{env}} \uplus \Delta_{\text{arg}}$ and $\Gamma_{\text{env}} \uplus \Gamma_{\text{arg}}$ respectively. These assignments must be chosen so that all of the free value variables of the term are contained in Γ'_{env} and furthermore, all of the free type variables of the term and the value environment must be contained in Δ'_{env}.

The chief technical difficulty in formulating these rules is that we need *two* type assignments, Γ'_{env} and Γ''_{env}, to describe the value environment of the closure, depending upon the context. The type assignment Γ'_{env} is constructed from the context $\Delta_{\text{env}};\Delta_{\text{arg}};\Gamma_{\text{env}};\Gamma_{\text{arg}}$ and is used to build the environment e_{ve} in the context in which the closure is constructed. The type assignment Γ''_{env} is obtained from Γ'_{env} via the translation $\Delta'_{\text{env}};\emptyset \rhd \Gamma'_{\text{env}} \rightsquigarrow \Gamma''_{\text{env}}$ and corresponds to the type of the value environment in the context of the closure itself. This ensures that the code of the closure is closed since the type ascribed to the value environment argument does not refer to free type variables in the context where the closure was created.

The type correctness of the translation is proved by induction on the derivation of the translation.

Theorem 4 *If $\Delta_{\text{env}};\Delta_{\text{arg}};\Gamma_{\text{env}};\Gamma_{\text{arg}} \rhd e \rightsquigarrow e'$ and $\Delta_{\text{env}} \uplus \Delta_{\text{arg}};\Gamma_{\text{env}} \uplus \Gamma_{\text{arg}} \vdash e : \sigma$, then $\{t_{\text{te}}{::}|\Delta_{\text{env}}|\} \uplus \Delta_{\text{arg}};\{x_{\text{ve}}{:}|\Gamma'_{\text{env}}|\} \uplus \Gamma'_{\text{arg}} \vdash e' : \sigma'$ where $\Delta_{\text{env}};\Delta_{\text{arg}} \rhd \sigma \rightsquigarrow \sigma'$, $\Delta_{\text{env}};\Delta_{\text{arg}} \rhd \Gamma_{\text{env}} \rightsquigarrow \Gamma'_{\text{env}}$, and $\Delta_{\text{env}};\Delta_{\text{arg}} \rhd \Gamma_{\text{arg}} \rightsquigarrow \Gamma'_{\text{arg}}$.*

The correctness of the translation may be established using an argument similar to that given for the simply-typed case. The restriction to predicative polymorphism significantly simplifies the proof.

5.2 Closure Representation

We now turn to the representation of closures for the polymorphic language.

The target language for polymorphic closure representation, called $\lambda^{\forall,\exists}$, is defined as follows:

$$\begin{array}{lrl} \textit{Kinds} & \kappa ::= & k \mid \Omega \mid \langle\kappa_1 \times \ldots \times \kappa_n\rangle \\ \textit{Types} & \sigma ::= & b \mid t \mid \langle\sigma_1 \times \ldots \times \sigma_n\rangle \mid \langle\sigma_1,\ldots,\sigma_n\rangle \mid \pi_i\ \sigma \mid \\ & & \forall t{::}\kappa.\sigma \mid \sigma_1 \Rightarrow \sigma_2 \mid \sigma_1 \to \sigma_2 \mid \exists t{::}\kappa.\sigma \mid \exists k.\sigma \\ \textit{Exp's} & e ::= & x \mid c \mid \lambda x{:}\sigma.e \mid e_1\ e_2 \mid \Lambda t{::}\kappa.e \mid e\ \sigma \mid \\ & & \langle e_1,\ldots,e_n\rangle \mid \pi_i\ e \mid \\ & & \texttt{pack}\ \sigma\ \texttt{with}\ e\ \texttt{as}\ \sigma' \mid \\ & & \texttt{open}\ e\ \texttt{as}\ t{::}\kappa\ \texttt{with}\ x{:}\sigma\ \texttt{in}\ e' \\ & & \texttt{pack}\ \kappa\ \texttt{with}\ e\ \texttt{as}\ \sigma \mid \\ & & \texttt{open}\ e\ \texttt{as}\ k\ \texttt{with}\ x{:}\sigma\ \texttt{in}\ e' \end{array}$$

Our translation of function types involves existential quantifiers. Since function types can instantiate a polymorphic type in the source language, we need to be able to instantiate polymorphic types with existentials in the target language. As a consequence, the target language must be *impredicative*. To simplify the language, we provide general abstractions (λ and Λ), instead of code types that abstract more than one argument at a time.

Since we shall have limited need of existential kinds, we must introduce kind variables k into the language, with corresponding kind contexts and judgements. A kind context $\mathcal{K}$ is simply a sequence of kind variables $\{k_1,\ldots,k_n\}$, $(n \geq 0)$. The typing judgements of the language are as follows:

$\mathcal{K};\Delta \vdash \sigma :: \kappa$	σ has kind κ.
$\mathcal{K};\Delta \vdash \sigma_1 \equiv \sigma_2 :: \kappa$	σ_1 and σ_2 are equal types of kind κ.
$\mathcal{K};\Delta;\Gamma \vdash e : \sigma$	e has type σ.

The formation rules, definitional equality rules, and typing rules are standard except that values of polymorphic type $\forall t{::}\kappa.\sigma$ may be coerced to the special type $\sigma' \Rightarrow \sigma[\sigma'/t]$, where σ' is a type of kind κ, as described in the Section 4. The details of the typing rules are found in the companion technical report [21].

We define the closure representation stage as a type-directed translation from $\lambda^{\forall,cl}$ to $\lambda^{\forall,\exists}$. We begin by defining a translation from source constructors and types to target type as follows:

$$\begin{array}{rcl} |t| & = & t \\ |b| & = & b \\ |\langle\sigma_1,\ldots,\sigma_n\rangle| & = & \langle|\sigma_1|,\ldots,|\sigma_n|\rangle \\ |\pi_i\sigma| & = & \pi_i|\sigma| \\ |\langle\sigma_1 \times \ldots \times \sigma_n\rangle| & = & \langle|\sigma_1| \times \ldots \times |\sigma_n|\rangle \\ |\texttt{vcode}(t{::}\kappa,\sigma_{\text{ve}},\sigma_1,\sigma_2)| & = & \forall t{::}\kappa.|\sigma_{\text{ve}}| \to |\sigma_1| \to |\sigma_2| \\ |\texttt{tcode}(t{::}\kappa,\sigma_{\text{ve}},s{::}\kappa',\sigma_2)| & = & \forall t{::}\kappa.|\sigma_{\text{ve}}| \to \forall s{::}\kappa'.|\sigma_2| \end{array}$$
$$|\sigma_1 \to \sigma_2| = \exists k.\exists t_0{::}\Omega.\exists t{::}k.\langle(t \Rightarrow t_0 \to |\sigma_1| \to |\sigma_2|) \times t_0\rangle$$
$$|\forall s{::}\kappa.\sigma_2| = \exists k.\exists t_0{::}\Omega.\exists t{::}k.\langle(t \Rightarrow t_0 \to \forall s{::}\kappa.|\sigma_2|) \times t_0\rangle$$

$$\frac{\{t_{\mathrm{te}}{::}\kappa_{\mathrm{te}}\};\{x_{\mathrm{ve}}{:}\sigma_{\mathrm{ve}}, x{:}\sigma_1\} \vdash e : \sigma_2}{\Delta;\Gamma \vdash \Lambda t_{\mathrm{te}}{::}\kappa_{\mathrm{te}}.\lambda x_{\mathrm{ve}}{:}\sigma_{\mathrm{ve}}.\lambda x{:}\sigma_1.e : \mathtt{vcode}(t_{\mathrm{te}}{::}\kappa_{\mathrm{te}}, \sigma_{\mathrm{ve}}, \sigma_1, \sigma_2)}$$

$$\frac{\{t_{\mathrm{te}}{::}\kappa_{\mathrm{te}}, t{::}\Omega\};\{x_{\mathrm{ve}}{:}\sigma_{\mathrm{ve}}\} \vdash e : \sigma}{\Delta;\Gamma \vdash \Lambda t_{\mathrm{te}}{::}\kappa_{\mathrm{te}}.\lambda x_{\mathrm{ve}}{:}\sigma_{\mathrm{ve}}.\Lambda t{::}\Omega.e : \mathtt{tcode}(t_{\mathrm{te}}{::}\kappa_{\mathrm{te}}, \sigma_{\mathrm{ve}}, t, \sigma)}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \mathtt{vcode}(t_{\mathrm{te}}{::}\kappa_{\mathrm{te}}, \sigma_{\mathrm{ve}}, \sigma_1, \sigma_2) \quad \Delta \vdash \tau : \kappa_{\mathrm{te}} \quad \Delta;\Gamma \vdash e_2 : \sigma_{\mathrm{ve}}[\tau/t_{\mathrm{te}}]}{\Delta;\Gamma \vdash \langle\!\langle e_1, \tau, e_2 \rangle\!\rangle : (\sigma_1 \to \sigma_2)[\tau/t_{\mathrm{te}}]}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \mathtt{tcode}(t_{\mathrm{te}}{::}\kappa_{\mathrm{te}}, \sigma_{\mathrm{ve}}, t, \sigma) \quad \Delta \vdash \tau : \kappa_{\mathrm{te}} \quad \Delta;\Gamma \vdash e_2 : \sigma_{\mathrm{ve}}[\tau/t_{\mathrm{te}}]}{\Delta;\Gamma \vdash \langle\!\langle e_1, \tau, e_2 \rangle\!\rangle : (\forall t{::}\Omega.\tau_2)[\tau/t_{\mathrm{te}}]}$$

Figure 5: Typing Rules for Code and Closures of $\lambda^{\forall,cl}$

$$\Delta_{\mathrm{env}};\Delta_{\mathrm{arg}} \rhd b \leadsto b \qquad \{t_1{::}\Omega,\ldots,t_n{::}\Omega\};\Delta_{\mathrm{arg}} \rhd t_i \leadsto \pi_i\, t_{\mathrm{te}}$$

$$\Delta_{\mathrm{env}};\Delta_{\mathrm{arg}} \rhd t \leadsto t \quad (t \in \mathit{Dom}(\Delta_{\mathrm{arg}}))$$

$$\frac{\Delta_{\mathrm{env}};\Delta_{\mathrm{arg}} \rhd \sigma_1 \leadsto \sigma_1' \quad \Delta_{\mathrm{env}};\Delta_{\mathrm{arg}} \rhd \sigma_2 \leadsto \sigma_2'}{\Delta_{\mathrm{env}};\Delta_{\mathrm{arg}} \rhd \sigma_1 \to \sigma_2 \leadsto \sigma_1' \to \sigma_2'} \qquad \frac{\Delta_{\mathrm{env}};\Delta_{\mathrm{arg}} \uplus \{t{::}\Omega\} \rhd \sigma \leadsto \sigma'}{\Delta_{\mathrm{env}};\Delta_{\mathrm{arg}} \rhd \forall t{::}\kappa.\,\sigma \leadsto \forall t{::}\kappa.\,\sigma'}$$

$$\frac{\Delta_{\mathrm{env}};\Delta_{\mathrm{arg}} \rhd t_1' \leadsto \tau_1 \quad \cdots \quad \Delta_{\mathrm{env}};\Delta_{\mathrm{arg}} \rhd t_n' \leadsto \tau_n}{\Delta_{\mathrm{env}};\Delta_{\mathrm{arg}} \rhd \{t_1'{::}\Omega,\ldots,t_n'{::}\Omega\} \leadsto \langle \tau_1,\ldots,\tau_n \rangle}$$

$$\frac{\Delta_{\mathrm{env}};\Delta_{\mathrm{arg}} \rhd \sigma_1 \leadsto \sigma_1' \quad \cdots \quad \Delta_{\mathrm{env}};\Delta_{\mathrm{arg}} \rhd \sigma_n \leadsto \sigma_n'}{\Delta_{\mathrm{env}};\Delta_{\mathrm{arg}} \rhd \{x_1{:}\sigma_1,\ldots,x_n{:}\sigma_n\} \leadsto \{x_1{:}\sigma_1',\ldots,x_n{:}\sigma_n'\}}$$

Figure 6: Polymorphic Abstract Closure Conversion: Types and Type Assignments

$$(\mathit{const}) \quad \Delta_{\mathrm{env}};\Delta_{\mathrm{arg}};\Gamma_{\mathrm{env}};\Gamma_{\mathrm{arg}} \rhd c \leadsto c$$

$$(\mathit{env}) \quad \Delta_{\mathrm{env}};\Delta_{\mathrm{arg}};\{x_1{:}\sigma_1,\ldots,x_n{:}\sigma_n\};\Gamma_{\mathrm{arg}} \rhd x_i \leadsto \pi_i\, x_{\mathrm{ve}}$$

$$(\mathit{arg}) \quad \Delta_{\mathrm{env}};\Delta_{\mathrm{arg}};\Gamma_{\mathrm{env}};\Gamma_{\mathrm{arg}} \rhd x \leadsto x \qquad (x \in \mathit{Dom}(\Gamma_{\mathrm{arg}}))$$

$$(\mathit{abs}) \quad \frac{\begin{array}{c}\Delta_{\mathrm{env}};\Delta_{\mathrm{arg}};\Gamma_{\mathrm{env}};\Gamma_{\mathrm{arg}} \rhd \Delta_{\mathrm{env}}' \leadsto \tau_{\mathrm{te}} \qquad \Delta_{\mathrm{env}};\Delta_{\mathrm{arg}};\Gamma_{\mathrm{env}};\Gamma_{\mathrm{arg}} \rhd \Gamma_{\mathrm{env}}' \leadsto e_{\mathrm{ve}} \\ \Delta_{\mathrm{env}}';\emptyset \rhd \Gamma_{\mathrm{env}}' \leadsto \Gamma_{\mathrm{env}}'' \qquad \Delta_{\mathrm{env}}';\emptyset \rhd \sigma_1 \leadsto \sigma_1' \\ \Delta_{\mathrm{env}}';\emptyset;\Gamma_{\mathrm{env}}';\{x{:}\sigma_1\} \rhd e \leadsto e'\end{array}}{\Delta_{\mathrm{env}};\Delta_{\mathrm{arg}};\Gamma_{\mathrm{env}};\Gamma_{\mathrm{arg}} \rhd \lambda x{:}\sigma_1.e \leadsto \langle\!\langle \Lambda t_{\mathrm{te}}{::}|\Delta_{\mathrm{env}}'|.\lambda x_{\mathrm{ve}}{:}|\Gamma_{\mathrm{env}}''|.\lambda x{:}\sigma_1'.e', \tau_{\mathrm{te}}, e_{\mathrm{ve}} \rangle\!\rangle}$$

$$(\mathit{tabs}) \quad \frac{\begin{array}{c}\Delta_{\mathrm{env}};\Delta_{\mathrm{arg}};\Gamma_{\mathrm{env}};\Gamma_{\mathrm{arg}} \rhd \Delta_{\mathrm{env}}' \leadsto \tau_{\mathrm{te}} \qquad \Delta_{\mathrm{env}};\Delta_{\mathrm{arg}};\Gamma_{\mathrm{env}};\Gamma_{\mathrm{arg}} \rhd \Gamma_{\mathrm{env}}' \leadsto e_{\mathrm{ve}} \\ \Delta_{\mathrm{env}}';\emptyset \rhd \Gamma_{\mathrm{env}}' \leadsto \Gamma_{\mathrm{env}}'' \qquad \Delta_{\mathrm{env}}';\{t{::}\Omega\};\Gamma_{\mathrm{env}}';\emptyset \rhd e \leadsto e'\end{array}}{\Delta_{\mathrm{env}};\Delta_{\mathrm{arg}};\Gamma_{\mathrm{env}};\Gamma_{\mathrm{arg}} \rhd \Lambda t{::}\Omega.\,e \leadsto \langle\!\langle \Lambda t_{\mathrm{te}}{::}|\Delta_{\mathrm{env}}'|.\lambda x_{\mathrm{ve}}{:}|\Gamma_{\mathrm{env}}''|.\Lambda t{::}\Omega.\,e', \tau_{\mathrm{te}}, e_{\mathrm{ve}} \rangle\!\rangle}$$

$$(\mathit{app}) \quad \frac{\Delta_{\mathrm{env}};\Delta_{\mathrm{arg}};\Gamma_{\mathrm{env}};\Gamma_{\mathrm{arg}} \rhd e_1 \leadsto e_1' \quad \Delta_{\mathrm{env}};\Delta_{\mathrm{arg}};\Gamma_{\mathrm{env}};\Gamma_{\mathrm{arg}} \rhd e_2 \leadsto e_2'}{\Delta_{\mathrm{env}};\Delta_{\mathrm{arg}};\Gamma_{\mathrm{env}};\Gamma_{\mathrm{arg}} \rhd e_1\ e_2 \leadsto e_1'\ e_2'}$$

$$(\mathit{tapp}) \quad \frac{\Delta_{\mathrm{env}};\Delta_{\mathrm{arg}};\Gamma_{\mathrm{env}};\Gamma_{\mathrm{arg}} \rhd e \leadsto e' \quad \Delta_{\mathrm{env}};\Delta_{\mathrm{arg}} \rhd \tau \leadsto \tau'}{\Delta_{\mathrm{env}};\Delta_{\mathrm{arg}};\Gamma_{\mathrm{env}};\Gamma_{\mathrm{arg}} \rhd e\ \tau \leadsto e'\ \tau'}$$

$$(\mathit{context}) \quad \frac{\Delta_{\mathrm{env}};\Delta_{\mathrm{arg}};\Gamma_{\mathrm{env}};\Gamma_{\mathrm{arg}} \rhd x_i \leadsto e_i'}{\Delta_{\mathrm{env}};\Delta_{\mathrm{arg}};\Gamma_{\mathrm{env}};\Gamma_{\mathrm{arg}} \rhd \{x_1{:}\sigma_1,\ldots,x_n{:}\sigma_n\} \leadsto \langle e_1',\ldots,e_n' \rangle}$$

Figure 7: Polymorphic Abstract Closure Conversion: Terms

The code types are translated to the appropriate combination of target $\forall$ and $\rightarrow$ types. The translation of a function type abstracts the kind of the type environment, k, and the type of the value environment, t_0. The type environment t is paired with the code by using an existential type. Since the type of a code is instantiated by t, only the type environment of the closure can be given to the code. The code and the value environment are paired as in the simply-typed case. The translation of $\forall$ has the same structure as that of an arrow type.

The translation of expressions is summarized in Figure 8. The kind of the type environment, the type of the value environment, and the type environment are packed with the pair of the code and the value environment. In the translation of applications, the type environment is obtained from a closure by an `open` expression and the code and the value environment are obtained by projections. Then the type environment, the value environment, and the argument of application are passed to the code.

Type preservation is proved by induction on the structure of the translation derivation. The typing rules for $\sigma \Rightarrow \sigma'$ are essential to prove the cases for the translations of closures.

Theorem 5 *If* $\Delta; \Gamma \triangleright e : \sigma \rightsquigarrow e'$, *then* $\emptyset; \Delta; |\Gamma| \vdash e' : |\sigma|$.

The correctness of the translation can be proven using logical relations as in the simply typed case. However, the definition of the relations is more complicated because of the presence of polymorphic types and types of the form $\pi_i(\tau)$ in the language $\lambda^{\forall,cl}$. The relations and the proof appear in the companion technical report [21].

6 Summary and Conclusions

We have given a type-theoretic account of closure conversion by defining type-directed transformations for the simply-typed and polymorphic λ-calculi. The types used in the target languages of the translations may be characterized in a natural way based on the "closures as objects" principle. In both the simply-typed and polymorphic cases of closure representation, we used Pierce-Turner-style existentials to hide the representations of environments. In the polymorphic case, we took advantage of Harper-Lillibridge-style translucency to ensure that the same type environment is used to type both the code and the value environment of a closure.

Our translations preserve types, facilitating correctness proofs and composition with other type-based translations. Furthermore, our translations provide support for run-time type analysis and type-based, tag-free garbage collection.

We have put the ideas in this paper to practical use in two separate compilers for ML. One compiler is being used to study novel approaches to tag-free garbage collection. The other compiler, called TIL (Typed Intermediate Languages), provides a general framework for analyzing types at run time to support efficient data representations, efficient calling conventions, and "nearly" tag-free garbage collection in the presence of polymorphism [36]. Propagating types through closure conversion is necessary for both compilers so that types can be examined at run time.

We have found that propagating types through closure conversion (and other compilation phases) has an additional *engineering* benefit. In particular, we can automatically verify the type-integrity of each type-preserving phase in the compiler. Indeed, automatic type-checking has enabled us to isolate and eliminate various subtle bugs in TIL.

For simplicity, the current implementations of our compilers use only abstract closure conversion. However, both compilers extend this translation to avoid creation of closures for "known" functions in the style of Wand and Steckler [39]. In the future, we hope to use the closure representation phase described here to further expose closure handling operations to optimization.

7 Acknowledgements

We would like to thank Lars Birkedal, Andrzej Filinski, Mark Leone, Sue Older, Benjamin Pierce, Paul Steckler, David Tarditi, and the anonymous reviewers for their many helpful comments and suggestions.

References

[1] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[2] A. W. Appel and T. Jim. Continuation-passing, closure-passing style. In *ACM Symp. on Principles of Programming Languages*, 1989.

[3] D. E. Britton. Heap storage management for the programming language Pascal. Master's thesis, University of Arizona, 1975.

[4] L. Cardelli. The functional abstract machine. *Polymorphism*, 1(1), 1983.

[5] C. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. In *Functional Programming Languages and Computer Architecture*, pages 50–64, 1985.

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

[7] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, England, 1989.

[8] J. Gosling. Java intermediate bytecodes. In *ACM SIGPLAN Workshop on Intermediate Representations (IR'95)*, Jan. 1995.

[9] J. Hannan. A type system for closure conversion. In *The Workshop on Types for Program Analysis*, 1995.

[10] R. Harper and M. Lillibridge. Explicit polymorphism and CPS conversion. In *ACM Symp. on Principles of Programming Languages*, 1993.

[11] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules. In *ACM Symp. on Principles of Programming Languages*, pages 123–137, 1994.

[12] R. Harper, D. MacQueen, and R. Milner. Standard ML. Technical Report ECS–LFCS–86–2, Laboratory for the Foundations of Computer Science, Edinburgh University, Mar. 1986.

[13] R. Harper and J. C. Mitchell. On the type structure of Standard ML. *ACM Transaction on Programming Languages and Systems*, 15(2), 1993.

[14] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *ACM Symp. on Principles of Programming Languages*, pages 130–141, 1995.

[15] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Functional Programming Language and Computer Architecture*, LNCS 201, pages 190–203. Springer-Verlag, 1985.

$$(vcl)\ \frac{\Delta;\Gamma \rhd e : \texttt{vcode}(t_{\rm te}{::}\kappa_{\rm te}, \sigma'_{\rm ve}, \sigma'_1, \sigma'_2) \leadsto e' \quad \Delta;\Gamma \rhd e_{\rm ve} : \sigma_{\rm ve} \leadsto e'_{\rm ve}}{\Delta;\Gamma \rhd \langle\!\langle e, \tau, e_{\rm ve}\rangle\!\rangle : \sigma_1 \to \sigma_2 \leadsto \texttt{pack}\ \kappa_{\rm te}, |\sigma_{\rm ve}|, |\tau|\ \texttt{with}\ \langle e', e'_{\rm ve}\rangle\ \texttt{as}\ |\sigma_1 \to \sigma_2|}$$

$$(app)\ \frac{\Delta;\Gamma \rhd e_1 : \sigma_1 \to \sigma_2 \leadsto e'_1 \quad \Delta;\Gamma \rhd e_2 : \sigma_1 \leadsto e'_2}{\begin{array}{ll}\Delta;\Gamma \rhd e_1\ e_2 : \sigma_2 \leadsto & \texttt{open}\ e'_1\ \texttt{as}\ k_{\rm te}, t_{\rm ve}, t_{\rm te} \\ & \quad \texttt{with}\ y : \langle t_{\rm te} \Rightarrow t_{\rm ve} \to |\sigma_1| \to |\sigma_2| \times t_{\rm ve}\rangle \\ & \texttt{in}\ (\pi_1\, y)\ t_{\rm te}\ (\pi_2\, y)\ e'_2\end{array}}$$

$$(tcl)\ \frac{\Delta;\Gamma \rhd e : \texttt{tcode}(t_{\rm te}{::}\kappa_{\rm te}, \sigma'_{\rm ve}, t{::}\kappa, \sigma') \leadsto e' \quad \Delta;\Gamma \rhd e_{\rm ve} : \sigma_{\rm ve} \leadsto e'_{\rm ve}}{\Delta;\Gamma \rhd \langle\!\langle e, \tau, e_{\rm ve}\rangle\!\rangle : \forall t{::}\kappa.\sigma_2 \leadsto \texttt{pack}\ \kappa_{\rm te}, |\sigma_{\rm ve}|, |\tau|\ \texttt{with}\ \langle e', e'_{\rm ve}\rangle\ \texttt{as}\ |\forall t{::}\kappa.\sigma|}$$

$$(tapp)\ \frac{\Delta;\Gamma \rhd e : \forall t{::}\kappa.\sigma \leadsto e'}{\begin{array}{ll}\Delta;\Gamma \rhd e\ \tau : \sigma[\tau/t] \leadsto & \texttt{open}\ e'\ \texttt{as}\ k_{\rm te}, t_{\rm ve}, t_{\rm te} \\ & \quad \texttt{with}\ y : \langle t_{\rm te} \Rightarrow t_{\rm ve} \to \forall t{::}\kappa.|\sigma| \times t_{\rm ve}\rangle \\ & \texttt{in}\ (\pi_1\, y)\ t_{\rm te}\ (\pi_2\, y)\ |\tau|\end{array}}$$

Figure 8: Polymorphic Closure Representation

[16] R. Kelsey and P. Hudak. Realistic compilation by program translation –detailed summary –. In *ACM Symp. on Principles of Programming Languages*, pages 281–292, 1989.

[17] D. Kranz et al. Orbit: An optimizing compiler for Scheme. In *Proc. of the SIGPLAN '86 Symp. on Compiler Construction*, 1986.

[18] X. Leroy. Unboxed objects and polymorphic typing. In *ACM Symp. on Principles of Programming Languages*, 1992.

[19] X. Leroy. Manifest types, modules, and separate compilation. In *ACM Symp. on Principles of Programming Languages*, pages 109–122, 1994.

[20] D. MacQueen. Modules for Standard ML. In *Proc. ACM Conf. Lisp and Functional Programming*, pages 198–207, 1984. Revised version appears in [12].

[21] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. Technical Report CMU–CS–95–171, School of Computer Science, Carnegie Mellon University, July 1995.

[22] J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *ACM Transaction on Programming Languages and Systems*, 10(3), 1988.

[23] G. Morrisett, M. Felleisen, and R. Harper. Abstract models of memory management. In *Functional Programming Languages and Computer Architecture*, pages 66–77, June 1995.

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

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

[26] B. C. Pierce and D. N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, Apr. 1994. A preliminary version appeared in Principles of Programming Languages, 1993, and as University of Edinburgh technical report ECS-LFCS-92-225, under the title "Object-Oriented Programming Without Recursive Types".

[27] G. 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.

[28] U. S. Reddy. Objects as closures. In *Proc. ACM Conf. Lisp and Functional Programming*, 1988.

[29] J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the Annual ACM Conference*, pages 717–740, 1972.

[30] Z. Shao and A. W. Appel. Space-efficient closure representations. In *Proc. ACM Conf. Lisp and Functional Programming*, 1994.

[31] Z. Shao and A. W. Appel. A type-based compiler for Standard ML. In *Programming Language Design and Its implemenation*, pages 116–129, 1995.

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

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

[34] G. L. Steele Jr. Rabbit: A compiler for Scheme. Master's thesis, MIT, 1978.

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

[36] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. Technical report, School of Computer Science, Carnegie Mellon University, Oct. 1995. To appear.

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

[38] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *14th ACM Symposium on Operating Systems Principles*, Dec. 1993.

[39] M. Wand and P. Steckler. Selective and lightweight closure conversion. In *ACM Symp. on Principles of Programming Languages*, 1994.