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

We study the typing properties of *closure conversion* for simply-typed and polymorphic λ-calculi. Unlike most accounts of closure conversion, which only treat the untyped λ-calculus, we translate well-typed source programs to well-typed target programs. This allows later compiler phases to take advantage of types for representation analysis and tag-free garbage collection, and it facilitates correctness proofs. Our account of closure conversion for the simply-typed language takes advantage of a simple model of objects by mapping closures to *existentials*. Closure conversion for the polymorphic language requires additional type machinery, namely *translucency* in the style of Harper and Lillibridge's module calculus, to express the type of a closure.

1 Introduction

Closure conversion [30, 35, 6, 16, 15, 2, 38, 8] 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 is called a "closure", a data structure containing pure code and a representation of its environment.

A critical decision in closure conversion is the choice of representation of the environment as a data structure — for example, whether to use a "flat", "linked", or hybrid representation. This decision is influenced by a desire to minimize closure creation time, the space consumed by an environment, and the time to access a given variable in an environment [38, 31]. An important property of closure conversion is that the representation of the environment is *private* to the closure, and is not visible from the outside. This affords considerable flexibility in the representation of environments and is thus exploited to good advantage by Shao and Appel [31] and Wand and Steckler [38].

Most accounts consider closure conversion as a transformation to *untyped* terms, irrespective of whether or not the source term is typed [35, 16, 2, 38]. 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, 32] uses types to determine procedure calling conventions, and Ohori's record compilation [26] uses a representation of types at run time to access components of a record. In current compilers, these phases must occur *before* closure conversion because the output of closure conversion is untyped. Compilation strategies for polymorphic languages, such as those proposed by Morrison *et al.* [25] and Harper and Morrisett [13], rely on analyzing types at run time to support unboxed representations and non-parametric operators, including printing and structural equality. Tag-free garbage collection [4, 37, 24] for both monomorphic and polymorphic programming languages also relies upon types at run time to determine the size and the pointers of objects. To support any of these implementation strategies, it is necessary to propagate type information *through* closure conversion and into the generated code. Consequently, the purpose of this paper is to show how closure conversion can be formulated as a *type-preserving transform.*

We are therefore interested in *type-based transformations* as a basis for compiling polymorphic languages. The crucial idea is to define a compiler as a series of transformations on both the program and its type, possibly relying on type information to guide the transformation itself. Each stage of the compiler is thus viewed as a type-preserving translation between typed intermediate languages. Examples of such translations are given by Leroy [18], Ohori [26], Harper and Lillibridge [9], and Harper and Morrisett [13]. In addition to the practical advantages of propagating type information through the stages of a compiler, type-directed translation also facilitates correctness proofs by defining the invariants of the transformation as a type-indexed family of *logical relations* [36, 7, 28, 33, 34].

We describe closure conversion in two stages. The first stage, *abstract closure conversion*, is a type-based translation from the source language into a target language with explicit closures. The translation is described as a deductive system in which the representation of the environment may be chosen independently for each closure. In this way vari-

[*]This research was performed while the author was visiting the Fox Project at Carnegie Mellon University.

To appear POPL '96.

ous environment representations, such as those used by the CAM [6] and the FAM [5], as well as hybrid strategies, such as those suggested by Shao and Appel [31] can be explained in a uniform framework.

The second stage, *closure representation*, is a type-based translation in which the implementation of closures is determined. The main idea is to represent *closures as objects* (in contrast to the proposed representation of *objects as closures* [29]). Following Pierce and Turner [27] we consider objects to be packages of existential type consisting of a single method (the code part of the closure) together with a single instance variable (the environment part) whose type (the environment representation) is held abstract. This captures the critical "privacy" property of environment representations for closures. In the simply-typed case we make direct use of Pierce and Turner's model of objects. In the polymorphic case we must in addition exploit the notion of *translucency* [10] (or *manifest types* [19]) to express the type of a polymorphic closure.

The correctness of both the abstract closure conversion and the closure representation stages are proved 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 which we may conclude that a closed program and its compilation evaluate to the same result. Due to a lack of space, the proofs of correctness are omitted here. However, full details are given in the companion technical report [22].

Closure conversion is discussed in descriptions of various functional language compilers [35, 16, 3, 2, 31]. It is closely related to λ-lifting [14] 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" [31], and thus choosing a good environment representation is an important part of compilation. Wand and Steckler [38] 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 [8] re-casts Wand's work into a typed setting, and provides correctness proofs for Wand's optimizations. Hannan's translation is given, like ours, as a deductive system, but like λ-lifting, he does not consider the important issue of environment representation (preferring an abstract account), 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.

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 and the other compiler provides a general framework for analyzing types at run time to determine the shapes of objects. Propagating types through closure conversion is necessary for both compilers so that types can be examined at run time. We have also found that typed closure conversion, along with our other type-preserving translations, made it possible to find and eliminate compiler bugs since we can automatically type-check the output of each compiler phase. Some compilers for ML based on representation analysis [18, 32] also propagate type information through closure conversion. However, their information is not enough to type-check the resulting programs because polymorphism is implemented by coercions and all polymorphic types are represented by a single type.

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 may be 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`, but not `z`. We may eliminate the references to these variables from the body of `f` by abstracting on 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.

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 applying it simultaneously to 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 $\tau_{\text{ve}} \to \tau_1 \to \tau_2$, where τ_{ve} is the type of the environment env, the closure as a whole would have type $(\tau_{\text{ve}} \to \tau_1 \to \tau_2) \times \tau_{\text{ve}}$, showing the type of the environment explicitly. That violates the "privacy" of the environment representation. As a result, using this translation on a well-typed source program will not, in general, result in a well-typed target program. For example, consider the following ML source program with type int → int:

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

Performing the translation above yields:

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

This program fails to type-check because the then-arm of the if-expression has type ({y:int} → int → int)×{y:int} while the else-arm has type ({} → int → int) × {}.

In order to preserve types in the target language, the representation of the environment may be hidden using existential types [23]. Figure 1 gives the typing rules for existentials. A pack operation pairs a type τ with a value e as an existential, holding τ abstract as a type variable, t. An open operation takes a package e_1 and opens it, binding the abstract type to t and the value of the package to x within the scope of e_2. The abstract type t is constrained so that it cannot leave the scope of the open construct, hence the restriction that t not appear in the free type variables of σ.

Using pack, we can hide the type of the environment for a closure value as follows:

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

A closure of type $\tau_1 \to \tau_2$ is represented as a package of type $\exists t_{\text{ve}}.(t_{\text{ve}} \to \tau_1 \to \tau_2) \times t_{\text{ve}}$ where the type of the environment (τ_{ve}) is held abstract as t_{ve}. Under this translation, the example above would be translated to:

```
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
```

Since the types of the arms of the if-expression agree, the target code is well-typed with type $\exists t_{\text{ve}}.(t_{\text{ve}} \to \text{int} \to \text{int}) \times t_{\text{ve}}$.

An application e e' is correspondingly translated to the expression

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

which opens the package, extracts the code and environment, and applies the code to the environment and the argument.

This representation of closures bears a striking resemblance to the model of objects suggested by Pierce and Turner [27]. In their model an object has a type of the form $\exists t.t \times \tau[t]$, where t is the type of the instance variable(s) and $\tau[t]$ is the type of the method(s). According to the foregoing account, closures may be thought of as objects with one instance variable (the environment) and one method (the code).

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. The conversion is described in increasing detail by three stages: The first stage, *abstract closure conversion*, converts each function to a closure but holds the representation of the closure abstract. To simplify the presentation, some freedom is allowed in the construction of environments, but no shared environments are used. The second stage, *environment sharing*, adds more structure to the translation thereby allowing environments to be shared. The third stage, *closure representation*, makes the representation of closures explicit through the use of translucent sums. Each stage is defined as a type-directed translation and the correctness of the translations is established using logical relations.

The syntax of the source language is defined as follows:

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

Types (τ) consist of base types (b) and function types ($\to$)[1]. Expressions (e) 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 are distinct. 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. We define the dynamic semantics of the language using a judgement of the form $e \hookrightarrow v$ (e evaluates to v). The judgement is derived from the following standard inference rules for call-by-value evaluation:

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

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

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

Figure 1: Typing Rules for Existentials

3.1 Abstract Closure Conversion

The target language for abstract closure conversion, λ^{cl} , is defined as follows:

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

In the introduction we informally presented closures as partial applications. As noted, we wish to delay this partial application until the closure is applied to an argument, so that the code and environment remain separate and the code can be shared among each instantiation of the closure. Therefore, in this account of closure conversion, we represent the delayed partial application as an abstract closure of the form $\langle\!\langle e, e_{\text{ve}} \rangle\!\rangle$ where e is the code and e_{ve} is the environment. This allows us to distinguish between delayed partial applications (closures) and closure application ($e_1\ e_2$). Code expressions, $\lambda x_{\text{ve}}{:}\tau_{\text{ve}}.\lambda x{:}\tau_1.e$, are a restricted form of closed λ-expressions that abstract both an environment (x_{ve}) and an argument (x). The types of code expressions are also distinguished from the types of closures and are written as $\texttt{code}(\tau_{\text{ve}}, \tau_1, \tau_2)$ where τ_{ve}, τ_1, and τ_2 are the types of the environment, the argument, and the return value respectively.

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

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

$$\frac{\Gamma \vdash e : \texttt{code}(\tau_{\text{ve}}, \tau_1, \tau_2) \quad \Gamma \vdash e_{\text{ve}} : \tau_{\text{ve}}}{\Gamma \vdash \langle\!\langle e, e_{\text{ve}} \rangle\!\rangle : \tau_1 \to \tau_2}$$

Since we require code to be closed in order that it may be hoisted to the top level, only $x_{\text{ve}}{:}\tau_{\text{ve}}$ (the environment) and $x{:}\tau_1$ (the argument) can be assumed when typing the body of the code.

Evaluation of closures and application proceeds as follows:

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

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

We define abstract closure conversion as a type-directed translation from the source language to λ^{cl} in Figure 2. The translation is formulated as a deductive system with judgements of the form $\Gamma; x{:}\tau \rhd e \leadsto e'$, and $\Gamma; x{:}\tau \rhd \Gamma' \leadsto e'_{\text{ve}}$ where Γ and Γ' are source type assignments, τ is a source type, e is a source expression, and e' and e'_{ve} are target expressions. The variable x is considered as the current argument while the other free variables in a source expression should be in Γ and accessed through the current environment in the translation. The judgement $\Gamma; x{:}\tau \rhd e \leadsto e'$ asserts that e' is the translation of e under the assumption that $\Gamma \uplus \{x{:}\tau\} \vdash e : \tau'$ for some τ'. The judgement $\Gamma; x{:}\tau \rhd \Gamma' \leadsto e'_{\text{ve}}$ asserts that e'_{ve} is an expression that evaluates to the environment corresponding to Γ' under the assumption that each binding in Γ' occurs in $\Gamma \uplus \{x{:}\tau\}$. Note that the order of bindings in Γ is important, and thus it is considered to be a sequence and not a set.

In a translated expression, x_{ve} is always used to hold the current local environment. Consequently, the translation rule (*env*) maps a source variable x_i found in the i^{th} position of type assignment Γ to the i^{th} projection of the environment variable x_{ve}, while the rule (*arg*) translates the argument variable x to itself.

The translation of an abstraction produces a closure consisting of code and an environment. To construct the environment, we choose a type assignment Γ' such that $\Gamma; x'{:}\tau' \rhd \Gamma' \leadsto e_{\text{ve}}$ is derivable via the (*context*) rule and $\Gamma'; x{:}\tau \rhd e \leadsto e'$. These two constraints can be summarized by saying that every binding in Γ' can also be found in $\Gamma \uplus \{x'{:}\tau'\}$. In a more detailed formulation, Γ' would be obtained from $\Gamma \uplus \{x'{:}\tau'\}$ via the application of *strengthening* and *exchange* rules. Furthermore, Γ' is required to contain bindings for all of the free variables in the original function $\lambda x{:}\tau.\, e$. The environment itself is constructed via the (*context*) rule by translating each of the variables occurring in Γ' (namely $x_1, \cdots, x_n$) to the target expressions $e_1, \cdots, e_n$. The resulting expressions are placed in a tuple ($\langle e_1, \ldots, e_n \rangle$) to form the environment data structure of the closure. 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', and then we abstract the environment and argument, resulting in $\lambda x_{\text{ve}}{:}|\Gamma'|.\, \lambda x{:}\tau.\, e'$.

Using a dummy argument ($x{:}b$) 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* $\emptyset; x{:}b \rhd e \leadsto e'$, *then* $\emptyset \vdash e' : \tau$.

To prove the operational 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:

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

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:

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

$$(\mathit{const})\ \Gamma; x{:}\tau \rhd c \rightsquigarrow c \quad (\mathit{arg})\ \Gamma; x{:}\tau \rhd x \rightsquigarrow x \quad (\mathit{env})\ \{x_1{:}\tau_1, \ldots, x_n{:}\tau_n\}; x{:}\tau \rhd x_i \rightsquigarrow \pi_i(x_{\mathrm{ve}})$$

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

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

Figure 2: Simply-Typed Abstract Closure Conversion

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

This theorem and the definition of the relations imply that for a closed program with a base type, the results of evaluation of the original program and its translation are the same.

3.2 Sharing Environments

Some implementations of functional programming languages use environments with nested structures that may share some portions of the environment with other closures. Sharing environments decreases the amount of space consumed by a closure and decreases the time to construct the closure's environment. However, sharing can also require extra instructions to access a variable's binding in the environment. Furthermore, sharing environments naively can lead to space problems in the presence of a standard tracing garbage collector. In this section we extend our closure conversion to allow for but not require shared environments. We do so by adding extra structure to the typing contexts of the translation judgement and use this extra structure to guide the construction of [possibly] shared environments. We then show how the resulting translation subsumes a wide variety of environment representations used in practice.

In the previous section, translation judgements were of the form $\Gamma; x{:}\tau \rhd e \rightsquigarrow e'$ where Γ was a *flat* type assignment of the form $\{x_1{:}\tau_1, \ldots, x_n{:}\tau_n\}$. Here, we add extra structure to the translation judgement by using *nested* type assignments defined 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 is represented in the target language by type $|\Theta|$ where $|\{x{:}\tau\}| = \tau$ and $|\langle \Theta_1, \ldots, \Theta_m \rangle| = \langle |\Theta_1| \times \ldots \times |\Theta_m| \rangle$. Clearly, we can obtain a non-nested type assignment (Γ) from a nested type assignment (Θ) simply by dropping the extra structure. Hence, we consider Θ to represent a nested type assignment as well as its corresponding flat type assignment.

The relevant translation rules for closure conversion with nested environments are given in Figure 3. The other translation rules are the same as in Figure 2, replacing Γ with Θ.

The (*arg*) rule translates a nested type assignment consisting of only the current argument to the variable itself. The (*env*) rule gives us the current environment directly as x_{ve} allowing us to avoid creating a copy. This rule, coupled with the (*env-tuple*) rule allows us to construct shared environments as nested tuples. If we translate Θ to e under the type assignment Θ_i, then the (*subenv*) rule lets us translate Θ to $e[\pi_i(x_{\mathrm{ve}})/x_{\mathrm{ve}}]$ under a type assignment which contains Θ_i as the i^{th} component. Variable x is translated as a nested type assignment $\{x{:}\tau\}$ by (*var*).

Nested type assignments are flexible enough to represent various environment representations used in practice. For example, the Categorical Abstract Machine or CAM [6] uses linked lists to represent environments. This is reflected in our framework by restricting the shape of nested type assignments and by restricting the (*env-tuple*) rule to "cons" the current argument onto the current environment:

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

$$(\mathit{env\text{-}tuple}) \quad \Theta_c; x{:}\tau \rhd \langle x{:}\tau, \Theta_c \rangle \rightsquigarrow \langle x, x_{\mathrm{ve}} \rangle$$

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 [5] 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" [2] 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 [31] 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

Abstract closure conversion chooses an environment representation for each closure and makes the construction of closures explicit. However, abstract closure conversion makes

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

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

Figure 3: Simply-Typed Closure Conversion using Nested Environments

the extraction of the code and environment in an application implicit in the operational semantics. Ideally, these extraction operations should be explicit so that an optimizer can eliminate redundant projections. For instance, if the same closure is repeatedly applied to some arguments in a loop, we should be able to extract the code and environment of the closure one time, name these values, and then use these names within the loop.

We therefore define a target language $\lambda^{\exists}$ with existential types as follows:

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

This language is the same as λ^{cl} except for the addition of package values of type $\exists t.\tau$ that pair an abstract type (t) with a value of type τ. Function types ($\rightarrow$) are no longer necessary because they can be represented using other type constructors (namely $\exists$ and $\texttt{code}(\tau_{\mathrm{ve}}, \tau_1, \tau_2)$). In order to prevent the partial application of the code to its environment, we restrict applications to the form $e_1(e_2, e_3)$. The typing rules and evaluation of `pack` and `open` expressions are standard (see [23] and Figure 1).

We begin by defining a translation from λ^{cl} to $\lambda^{\exists}$ types, denoted $|\tau|$ and defined as follows:

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

The translation of an arrow type is a pair consisting of code and an environment, with the environment type (t_{ve}) held abstract using an existential.

The translation mapping λ^{cl} terms to $\lambda^{\exists}$ terms is summarized in Figure 4. The translation defines judgements of the form $\Gamma \triangleright e : \tau \leadsto e'$ 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. A closure is translated to a pair of the code and the environment packed with the type of the environment. The translation of an application extracts from a package the pair of a code and an environment and applies 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, writing:

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

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

Operational 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 relations and the proof of the operational correctness are found in the technical report [22].

4 Overview of Polymorphic Closure Conversion

Closure conversion for a language with ML-style (*i.e.*, predicative [12]), explicit polymorphism follows a similar pattern to the simply-typed case, but with the additional complication that we must account for free type variables as well as free value variables in the code of an abstraction, and both value abstractions (λ-terms) and type abstractions (Λ-terms) induce the creation of closures. 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. By abstracting both free type variables and free value variables, the code becomes closed and can be hoisted to the top level. The abstracted code is then "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 environments. Also, we need to 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 with the simply-typed case.

As a running example, consider the expression:

```
λx:t₁. (x:t₁, y:t₂, z:int)
```

of type $t_1 \rightarrow (t_1 \times t_2 \times \texttt{int})$ 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. After closure conversion, this expression is translated 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
```

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

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

Figure 4: Important Rules of Simply-Typed Closure Representation

The `code` abstracts type environment (`tenv`) and value environment (`venv`) arguments. The actual type environment, $\{t_1\texttt{=}t_1, t_2\texttt{=}t_2\}$, is a constructor record with 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 $\{\texttt{y:}t_2,\ \texttt{z:int}\}$. However, to keep the code closed so that it may be hoisted, all references to free type variables in the type of `venv` must come from `tenv`. Thus, we give `venv` the type $\{\texttt{y:\#}t_2\ \texttt{tenv},\ \texttt{z:int}\}$. Similarly, the code's argument `x` is given the type $\texttt{\#}t_1\ \texttt{tenv}$. Consequently, the `code` part of the closure is a closed expression of closed type σ, where

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

It is easy to check that the entire expression has type $t_1 \to (t_1 \times t_2 \times \texttt{int})$, and thus the type of the original function is preserved.

We must now translate the partial application of the `code` to it environments into a data structure. The structure must be "mixed-phased" because it needs to hold a type (the type environment) as well as values (the code and value environment). A first attempt is to represent the data structure as a package e, where

$$\begin{array}{l}e\ \texttt{=}\ \texttt{pack}\ \{t_1\texttt{=}t_1,\ t_2\texttt{=}t_2\}\ \texttt{with (code, \{y=y, z=z\})}\\ \qquad\quad \texttt{as}\ \exists t_{\mathrm{te}}\texttt{::}\kappa_{\mathrm{te}}\texttt{.}\sigma \times \tau_{\mathrm{ve}}\end{array}$$

and `code` is the code of the closure above and

$$\begin{array}{lcl}\kappa_{\mathrm{te}} & \texttt{=} & \{t_1\texttt{::}\Omega,\ t_2\texttt{::}\Omega\}\\ \tau_{\mathrm{ve}} & \texttt{=} & \{\texttt{y:\#}t_2\ t_{\mathrm{te}}\texttt{, z:int}\}\end{array}$$

It is easy to verify that e is well-typed under the typing rule for `pack`.

Unfortunately, there is a problem with this approach: an application of e to some argument $e' : t_1$ must open the package to extract the code, type, and value environments prior to the call:

$$\begin{array}{l}\texttt{open}\ e\ \texttt{as}\ t_{\mathrm{te}}\texttt{::}\kappa_{\mathrm{te}}\ \texttt{with z:}\sigma \times \tau_{\mathrm{ve}}\\ \texttt{in}\\ \qquad \texttt{(\#1 z)}\ t_{\mathrm{te}}\ \texttt{(\#2 z)}\ e'\\ \texttt{end}\end{array}$$

Although this is the "obvious" translation of application, it fails to be well-typed! The difficulty is that e' is of type t_1, whereas the expression $\texttt{(\#1 z)}\ t_{\mathrm{te}}\ \texttt{(\#2 z)}$ has type:

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

Since t_{te} is abstract, t_1 is not provably equivalent to $\texttt{\#}t_1\ t_{\mathrm{te}}$, and this translation of application fails to typecheck.

The problem is that existentials provide a certain kind of mixed-phase data structure where the type portion *must* be abstract. We can use this to hide representations but here, we need to know what the type environment actually is in order to determine the type of the closure. In short, we need a mixed-phase data structure that does not hide its type component.

This same problem has been encountered in the study of the ML-like module systems [11, 20, 21]. Recent solutions are based on the idea of *translucent sums* [10] or *manifest types* [19], which provide the power of both existentials (weak sums), and transparent sums (strong sums). By ascribing the translucent sum type

$$\exists t_{\mathrm{te}}\ \texttt{=}\{t_1\texttt{=}t_1\texttt{,}t_2\texttt{=}t_2\}\texttt{.}\sigma \times \tau_{\mathrm{ve}}$$

to the closure, the equation $t_{\mathrm{te}}\texttt{=}\{t_1\texttt{=}t_1\texttt{,}t_2\texttt{=}t_2\}$ is propagated into the scope of the abstraction so that in particular $\texttt{\#}t_1\ t_{\mathrm{te}}\ =\ \texttt{\#}t_1\ \{t_1\texttt{=}t_1\texttt{,}t_2\texttt{=}t_2\}\ =\ t_1$, and thus the translation of application is type correct.

The next step is to hide the representation of the value environment as we did in the simply-typed case. If we simply abstract the type τ_{ve} from the above type expression we obtain

$$\exists t_{\mathrm{te}}\ \texttt{=}\{t_1\texttt{=}t_1\texttt{,}t_2\texttt{=}t_2\}\texttt{.}\exists t_{\mathrm{ve}}\texttt{::}\Omega\texttt{.}\sigma \times t_{\mathrm{ve}}$$

where t_{ve} is the abstract type of the value environment. However, this fails to make type sense because we have abstracted the type of the value environment in the closure, but not the corresponding argument type of the code of the closure. The translation of application is ill-typed because the value environment has abstract type t_{ve}, but the domain type of $\texttt{(\#1 z)}\ t_{\mathrm{te}}$ is $\{\texttt{y:}t_2\texttt{, z:int}\}$. Since t_{ve} is abstract, these two types are considered distinct. In order to simultaneously abstract the type of the value environment and the corresponding argument type in the code, we need to replace both types with the same abstract type t_{ve}. To do this, we must show that the two types are equivalent. This can be accomplished by requiring that the formal type environment argument (`tenv`) is only instantiated with the type environment t_{te}. One way to achieve this is to perform the application of the code to t_{te}, but the goal of closure conversion is to *delay* such partial applications. An alternative approach is to use translucency again and *coerce* the code so that it has the type σ', where:

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

Adding the constraint $\texttt{tenv=}\ t_{\mathrm{te}}$ to the type of the code has the effect of performing the type application at the *type-level*, but delays the application at the *term-level*. Note that

σ' is a super-type of the original code type σ according to the rules of the translucent sum calculus. Consequently, the code remains the same (*i.e.*, closed) and can still be hoisted to the top level. In contrast, if we had performed the type application, the resulting code would not be closed (containing free references to t_{te}).

Since `tenv`$= t_{\text{te}}$ and $t_{\text{te}} = \{t_1\texttt{=}t_1, t_2\texttt{=}t_2\}$, it follows that `tenv`$=\{t_1\texttt{=}t_1, t_2\texttt{=}t_2\}$ and thus $(\texttt{\#}t_i\ \texttt{tenv}) = t_i$. Consequently, the data structure holding the components of the closure can be coerced to the equivalent type:

$$\exists t_{\text{te}} = \{t_1\texttt{=}t_1, t_2\texttt{=}t_2\}.\sigma'' \times \{\texttt{y}:t_1, \texttt{z}:\texttt{int}\}$$

where σ'' is

$$\sigma'' = \forall \texttt{tenv}= t_{\text{te}}::\kappa_{\text{te}}.\{\texttt{y}:t_1,\ \texttt{z}:\texttt{int}\}\rightarrow t_1 \rightarrow (t_1 \times t_2 \times \texttt{int})$$

Since this equivalent type makes no mention of the type environment t_{te} except in the constraint for `tenv`, we may drop the constraint on t_{te}, abstract the type of the value environment $(\{\texttt{y}:t_1,\texttt{z}:\texttt{int}\})$, and abstract the kind of the type environment κ_{te} to obtain the closure type:

$$\exists k_{\text{te}}.\exists t_{\text{te}}::k_{\text{te}}.\exists t_{\text{ve}}::\Omega.\sigma''' \times t_{\text{ve}}$$

where

$$\sigma''' = \forall \texttt{tenv}= t_{\text{te}}::k_{\text{te}}.t_{\text{ve}} \rightarrow t_1 \rightarrow (t_1 \times t_2 \times \texttt{int})$$

It is easy to derive a type-preserving translation of application corresponding to this representation of closures. We simply `open` all of the existentials, and pass the type environment, value environment, and argument to the code.

Careful consideration of the foregoing discussion reveals that only limited use is made of translucency. The equational constraint on t_{te} is dropped from the existential (to ensure privacy of environment representation), and the universally quantified variable `tenv` does not occur in the scope of the abstraction. This suggests that a substantially simpler mechanism than the full translucent sum calculus is more appropriate for closure conversion. Hence, we introduce a special type, written $\tau \Rightarrow \sigma$, 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 $\tau_1 \rightarrow \tau_2$ becomes

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

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 [27] for objects and by Harper and Lillibridge [10] for modules.

5 A Formal Account of Polymorphic Closure Conversion

In this section, we present closure conversion for the predicative subset of the second order λ-calculus. It has been argued that the predicative fragment captures the "essence" of ML-style polymorphism, since there is a stratification between monotypes (types not involving a quantifier) and polytypes, and instantiation of type variables is restricted to monotypes [12]. These restrictions make it easy to use logical relations to argue correctness in the same fashion as we did for the simply-typed λ-calculus.

The syntax of our source language $\lambda^{\forall}$ is defined as follows:

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

The type constructors (τ) are described by kinds (κ). There is only one kind (Ω) for $\lambda^{\forall}$, but subsequent languages have a richer kind structure, so we introduce kinds here for uniformity. Closed constructors of kind Ω correspond to a subset of types (the monotypes), in particular the types that do not include quantifiers. 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. A typing judgement is derived from the standard typing rules of the second-order λ-calculus (see for example [12, 13]).

5.1 Abstract Closures

As in the simply typed case, we break closure conversion into abstract closure conversion and closure representation stages[2]. The abstract closure conversion stage for $\lambda^{\forall}$ converts both λ-abstractions and Λ-abstractions into abstract closures consisting of code, a type environment and a value environment.

The syntax of the target language $\lambda^{\forall,cl}$ is defined as follows:

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

A product kind $\langle \kappa_1 \times \ldots \times \kappa_n \rangle$ is used to specify the shape of type environments just as a product type specifies the shape of value environments. Given constructors τ_i with kind κ_i, the constructor $\langle \tau_1, \ldots, \tau_n \rangle$ has kind $\langle \kappa_1 \times \ldots \times \kappa_n \rangle$ and is used as a type environment consisting of $\tau_1, \ldots, \tau_n$ in a translated program.

[2]The material on environment sharing carries over in a straightforward manner.

There are two types of codes: the code for ordinary abstraction, $\Lambda t_{\mathrm{te}}{::}\kappa_{\mathrm{te}}.\lambda x_{\mathrm{ve}}{:}\sigma_{\mathrm{ve}}.\lambda x{:}\sigma_1.e$, and the code for type abstraction, $\Lambda t_{\mathrm{te}}{::}\kappa_{\mathrm{te}}.\lambda x_{\mathrm{ve}}{:}\sigma_{\mathrm{ve}}.\Lambda t{::}\kappa.e$. Codes take a type environment, a value environment, and a type or value argument respectively. We introduce the types `vcode` and `tcode`, to distinguish the types of codes from the types of closures and to avoid partial applications of codes. Intuitively, they correspond to standard types as follows:

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

Only types excluding $\forall$, `vcode`, and `tcode` can be named as a constructor. An abstract closure $\langle\!\langle e_1, \tau, e_2 \rangle\!\rangle$ consists of a code e_1, a type environment constructor 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.

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. They consist of the equivalence rules for projections below as well as the standard rules for equivalence and congruence.

$$\begin{array}{c} \Delta \vdash \pi_i\langle \tau_1, \ldots, \tau_n \rangle \equiv \tau_i :: \kappa_i \\ \Delta \vdash \langle \pi_1(\tau), \ldots, \pi_n(\tau) \rangle \equiv \tau :: \langle \kappa_1 \times \ldots \times \kappa_n \rangle \end{array}$$

The typing rules for expressions are standard except for the rules for codes and closures. They 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 hoist code out of inner definitions to the top level.

Abstract closure conversion for $\lambda^{\forall}$ is formulated as a type-directed translation to $\lambda^{\forall,cl}$ by the deductive system in Figures 6 and 7. The judgement $\Delta_{\mathrm{env}}; \Delta_{\mathrm{arg}} \rhd \sigma \rightsquigarrow \sigma'$ 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 $\Delta_{\mathrm{env}}; \Delta_{\mathrm{arg}}; \Gamma_{\mathrm{env}}; \Gamma_{\mathrm{arg}} \rhd e \rightsquigarrow e'$ 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 $|\Delta_{\mathrm{env}}|$ and $|\Gamma_{\mathrm{env}}|$ respectively, 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 $\Delta_{\mathrm{env}} \uplus \Delta_{\mathrm{arg}}$ and $\Gamma_{\mathrm{env}} \uplus \Gamma_{\mathrm{arg}}$ respectively. These assignments must be chosen so that all of the free value variables of the term are contained in Γ'_{env} and further, all of the free type variables of the term and the value environment must be contained in Δ'_{env}.

The primary subtlety in 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. Γ'_{env} is constructed from the context $\Delta_{\mathrm{env}}; \Delta_{\mathrm{arg}}; \Gamma_{\mathrm{env}}; \Gamma_{\mathrm{arg}}$ and is used to build the environment e_{ve} in the context where we are constructing the closure. In contrast, Γ''_{env} is obtained from Γ'_{env} via the translation $\Delta'_{\mathrm{env}}; \emptyset \rhd \Gamma'_{\mathrm{env}} \rightsquigarrow \Gamma''_{\mathrm{env}}$ and corresponds to the type of the value environment in the context of the closure itself.

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

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

We can prove the operational correctness of the translation using logical relations in the same fashion as we did for the simply typed case because the source language is restricted to the predicative polymorphism.

5.2 Closure Representation

In this section we present closure representation for the second order language. We use types with existential *kinds* to abstract the representation of type environments and existential types to abstract the representation of value environments. Further, we introduce the type $\sigma \Rightarrow \sigma'$, derived from translucent types, to solve the typing problems discussed in the overview.

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

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

There is no distinction between types and constructors for $\lambda^{\forall,\exists}$ because type application is no longer restricted to just monotypes. $\lambda^{\forall,\exists}$ needs this impredicativity because some monotypes from the source language are translated into types with quantifiers. To simplify the language, we provide full abstractions (λ and Λ) instead of codes which abstract more than one argument at a time.

To provide types with existential kinds, we need to introduce kind variables k and kind contexts for the typing judgements of $\lambda^{\forall,\exists}$. 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 consist of the following:

$\mathcal{K}; \Delta \vdash \sigma :: \kappa$	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.

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

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

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

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

Figure 5: Typing Rules for Code and Closures

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 7: Polymorphic Abstract Closure Conversion: Terms

The definition of the formation rules, definitional equality of types, and typing rules are standard except that polymorphic type $\forall t :: \kappa.\sigma$ can be instantiated to $\tau \Rightarrow \sigma[\tau/t]$ for type τ with kind κ as described in the overview. The details of the typing rules are found in the technical report [22].

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:

$$\begin{array}{rcl}
|t| & = & t \\
|b| & = & b \\
|\langle\sigma_1,\ldots,\sigma_n\rangle| & = & \langle|\sigma_1|,\ldots,|\sigma_n|\rangle \\
|\pi_i\sigma| & = & \pi_i|\sigma| \\
|\langle\sigma_1 \times \ldots \times \sigma_n\rangle| & = & \langle|\sigma_1| \times \ldots \times |\sigma_n|\rangle \\
|\mathtt{vcode}(t{::}\kappa,\sigma_{\mathrm{ve}},\sigma_1,\sigma_2)| & = & \forall t{::}\kappa.|\sigma_{\mathrm{ve}}| \to |\sigma_1| \to |\sigma_2| \\
|\mathtt{tcode}(t{::}\kappa,\sigma_{\mathrm{ve}},s{::}\kappa',\sigma_2)| & = & \forall t{::}\kappa.|\sigma_{\mathrm{ve}}| \to \forall s{::}\kappa'.|\sigma_2| \\
\multicolumn{3}{c}{|\sigma_1 \to \sigma_2| = \exists k.\exists t{::}k.\exists t_0{::}\Omega.\langle(t \Rightarrow t_0 \to |\sigma_1| \to |\sigma_2|) \times t_0\rangle} \\
\multicolumn{3}{c}{|\forall s{::}\kappa.\sigma_2| = \exists k.\exists t{::}k.\exists t_0{::}\Omega.\langle(t \Rightarrow t_0 \to \forall s{::}\kappa.|\sigma_2|) \times t_0\rangle}
\end{array}$$

The code types are translated to the corresponding types described in the previous section. 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 environment, and the type of the value 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.

The type correctness of the translation is proven by induction on the derivation of the translation. The typing rules for $\sigma \Rightarrow \sigma'$ are essential to prove the cases for the translations of closures.

Theorem 5 (Type Correctness) *If* $\Delta;\Gamma \rhd e : \sigma \leadsto e'$, *then* $\emptyset;\Delta;|\Gamma| \vdash e' : |\sigma|$.

The operational 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 $\pi_i(\sigma)$ in the language $\lambda^{\forall,cl}$. The relations and the proof appear in the technical report [22].

6 Summary and Conclusions

We have presented a type-theoretic account of closure conversion for the simply-typed and polymorphic λ-calculi. Our translations are unique in that they map well-typed source terms to well-typed target terms. This facilitates correctness proofs, allows other type-directed transforms such as CPS conversion or unboxing to be applied after closure conversion, and supports run-time examination of types for 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 and the other compiler provides a general framework for analyzing types at run time to determine the shapes of objects. Propagating types through closure conversion is necessary for both compilers so that types can be examined at run time. For simplicity, the current implementations of our compilers use abstract closure conversion. As many compilers, our compilers avoid creation of closures for known-functions. Such optimization does not introduce any problem.

Lightweight closure conversion proposed by Wand and Steckler [38] may also be formulated as type-preserving translation as our closure conversions. However, the definition of translation may become complicated because the type of the closure as well as the translation of abstraction depends on which free variables passed directly.

In some implementation, a closure is represented by folding code pointer into the environment record [2]. Such representation may be formulated as abstract closure conversion with some modifications. However, closure representation for such closures seems to need more powerful type system.

7 Acknowledgements

We would like to thank Andrzej Filinski, Mark Leone, Benjamin Pierce, and David Tarditi for their comments and suggestions.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J.Lévy. Explicit substitutions. In *ACM Symp. on Principles of Programming Languages*, 1990.

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

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

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

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

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

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

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

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

[10] 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.

[11] 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.

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

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

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

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

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

Figure 8: Closure Representation

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

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

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

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

[17] P. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.

[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 [11].

[21] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[22] 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.

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

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

[25] 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.

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

[27] 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".

[28] 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.

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

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

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

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

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

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

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

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

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