Typed Closure Conversion for Recursively-Defined Functions (Extended Abstract)

Greg Morrisett [1]

Cornell University

Robert Harper [2]

Carnegie Mellon University

Abstract

Much recent work on the compilation of statically typed languages such as ML relies on the propagation of type information from source to object code in order to increase the reliability and maintainabilty of the compiler itself and to improve the efficiency and verifiability of generated code. To achieve this the program transformations performed by a compiler must be cast as type-preserving translations between typed intermediate languages. In earlier work with Minamide we studied one important compiler transformation, closure conversion, for the case of pure simply-typed and polymorphic λ-calculus. Here we extend the treatment of simply-typed closure conversion to account for recursively-defined functions such as are found in ML. We consider three main approaches, one based on a recursive code construct, one based on a self-referential data structure, and one based on recursive types. We discuss their relative advantages and disadvantages, and sketch correctness proofs for these transformations based on the method of logical relations.

[1] This material is based on work supported in part by the AFOSR grant F49620-97-1-0013 and ARPA/RADC grant F30602-96-1-0317. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not reflect the views of these agencies.

[2] This research was sponsored 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. 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 the Advanced Research Projects Agency or the U.S. Government.

1 Introduction

Closure conversion is a critical program transformation for higher-order languages that eliminates lexically nested, first-class functions or procedures. In particular, closure conversion translates each function definition f into a *closure* – a data structure consisting of a pointer to closed *code* and another data structure which represents the *environment* or context of the function. The code abstracts the arguments of f as well as the free variables of f, and the environment provides the values for the free variables of f. Function application is translated to a sequence which invokes the code of the function's closure on the environment of the closure and the arguments. Since the code is closed and separated from the data which it manipulates, it may be defined at the top-level and shared by all closures that are instances of the function. In this respect, closure conversion reifies meta-level constructs (*i.e.*, closures and environments) as object-level constructs (*i.e.*, code and tuples) in the same fashion that conversion into continuation-passing style reifies meta-level continuations as object-level functions.

As an example, the source level expression

$$
\begin{array}{l}
\texttt{let}\ \begin{array}[t]{ll}
f & = \lambda x.\lambda y.\lambda z.x + y + z \\
f' & = f\,3 \\
f'' & = f'\,4
\end{array} \\
\texttt{in} \\
\quad f''\,5 \\
\texttt{end}
\end{array}
$$

might be closure-converted to the target language expression

$$
\begin{array}{l}
\texttt{let}\ \begin{array}[t]{ll}
z_{\mathrm{code}} & = \lambda[e, z].(\pi_2\, e) + (\pi_1\, e) + z \\
y_{\mathrm{code}} & = \lambda[e, y].\langle z_{\mathrm{code}}, \langle y, \pi_1\, e, \rangle\rangle \\
x_{\mathrm{code}} & = \lambda[e, x].\langle y_{\mathrm{code}}, \langle x\rangle\rangle \\
f & = \langle x_{\mathrm{code}}, \langle\rangle\rangle \\
f' & = (\pi_1\, f)\, [\pi_2\, f, 3] \\
f'' & = (\pi_1\, f')\, [\pi_2\, f', 4]
\end{array} \\
\texttt{in} \\
\quad (\pi_1\, f'')\, [\pi_2\, f'', 5] \\
\texttt{end}
\end{array}
$$

The function f becomes a pair of the code x_{code} and an empty environment. The definition of f' invokes f by calling the code of f (x_{code}), passing to it its environment and the argument 3. The code builds a new closure, containing the code y_{code} and the environment consisting of the value bound to x ($i.e.$, 3). Hence, f' becomes bound to the value $\langle y_{\mathrm{code}}, \langle 3\rangle\rangle$. The definition of f'' invokes f' by calling the code y_{code}, passing to it its environment and the argument 4. The code builds a new closure, containing the code z_{code} and the environment consisting of the values of y ($i.e.$, 4) and x ($i.e.$, 3). Note that the value of x is obtained from the environment e. Hence, f'' becomes bound to the value $\langle z_{\mathrm{code}}, \langle 4, 3\rangle\rangle$. The body of the `let` invokes this closure on the argument 5. Hence, within the definition of z_{code}, e is bound to $\langle 4, 3\rangle$ and z is bound to 5. The body of this code computes the value $3 + 4 + 5 = 12$.

In earlier work with Minamide [6] we considered the question of how to perform closure conversion in a typed setting. We sought to relate the type of a program after closure conversion to its type prior to closure conversion. The key observation is that the naïve approach to closure conversion sketched above does not, in general, yield a well-typed term. For instance, consider the following source expression:

$$\texttt{if}\ c\ \texttt{then}\ \lambda x{:}\mathsf{int}.x + a + b\ \texttt{else}\ \lambda z{:}\mathsf{int}.z$$

Under the typing assumptions $c{:}\mathsf{bool}$, $a{:}\mathsf{int}$, and $b{:}\mathsf{int}$, this expression may be assigned the type $\mathsf{int} \to \mathsf{int}$. A typical closure conversion algorithm yields the following translation:

$$\begin{array}{l} \texttt{if}\ c\ \texttt{then}\ \langle \lambda[e{:}\langle \mathsf{int} \times \mathsf{int}\rangle, x{:}\mathsf{int}].x + \pi_1\, e + \pi_2\, e, \langle a, b\rangle\rangle \\ \texttt{else}\ \langle \lambda e{:}\langle\,\rangle, z{:}\mathsf{int}.z, \langle\rangle\rangle \end{array}$$

But this term is not well-typed in a conventional typed λ-calculus since the closure in the `then` clause has type $\langle([\langle \mathsf{int} \times \mathsf{int}\rangle, \mathsf{int}] \to \mathsf{int}) \times \langle \mathsf{int} \times \mathsf{int}\rangle\rangle$ whereas the closure in the `else` clause has type $\langle([\langle\,\rangle, \mathsf{int}] \to \mathsf{int}) \times \langle\,\rangle\rangle$. The issue is that, at the source level, $\mathsf{int} \to \mathsf{int}$ hides the type of the environment, but at the target level, the type of the environment is exposed in the type of the closure.

This problem of representation exposure may be avoided by using an existential type [7] to hide the type of the environment. This ensures that all closures arising from a given source language type have the same type after closure conversion. Specifically, source functions of type $\tau_1 \to \tau_2$ are translated to target closures with type $\exists\alpha.\langle([\alpha, \tau_1] \to \tau_2) \times \alpha\rangle$. This translation is closely related to Pierce and Turner's type system for objects [8] — a function is interpreted as an object with one method (the code) and one instance variable (the environment) using their existential type discipline for simple objects.

The present work is concerned with the extension of our previous work to account for recursively-defined functions. This generalization introduces two significant complications. First, several different approaches are available, and none dominates the others in all respects. We consider here three translations: one based on recursive code, one based on a self-referential data structure, and one based on recursive types. Second, the correctness proofs given by Minamide, Morrisett, and Harper based on logical relations must be extended to account for recursion. The general idea in each case is to relate the finite approximinants of a recursive function to a corresponding finite approximant of the target code. In the case of recursive code and self-referential data structures, the finite approximants are the finite unrollings of the recursive code or self-referential structure. In the case of recursive types we make use of the syntactic minimal invariant associated with a recursive type [9,5,4] to define the finite approximants of a value.

In the rest of this abstract, we sketch the original simply-typed closure conversion algorithm and the three translations mentioned above for recursive functions.

2 Simply-Typed Closure Conversion

We formalize closure conversion for the simply-typed lambda calculus by defining the syntax, static semantics, and dynamic semantics for a source language ($\lambda^{\rightarrow}$) and a target language ($\lambda^{\exists}$), by giving a type translation $\mathcal{T}[\![\ ]\!]$ from $\lambda^{\rightarrow}$ types to $\lambda^{\exists}$ types, and a term translation from well-formed $\lambda^{\rightarrow}$ terms to $\lambda^{\exists}$ terms.

The syntax for $\lambda^{\rightarrow}$ is:

$$\begin{aligned} (\text{types})\ \tau\ &::= \mathsf{b} \mid \tau_1 \rightarrow \tau_2 \\ (\text{terms})\ e\ &::=\ x \mid \mathsf{c} \mid \lambda x{:}\tau.e \mid e_1\, e_2 \end{aligned}$$

The language is a conventional simply-typed lambda calculus with a distinguished base type (b) inhabited by a set of constants, over which we use c to range. The static and dynamic semantics (not presented here) is entirely standard.

The syntax of our target language $\lambda^{\exists}$ is:

Types include products (for building both environments and closures), code ($\Rightarrow$) (for the code of closures) and type variables and existentially quantified types (for hiding the type of the environment of a closure). Our target language is *impredicative* in that type variables range over quantified as well as unquantified types.

$$\begin{array}{rl}
\text{(types)}\ \tau ::= & t \mid \mathtt{b} \mid \tau_1 \Rightarrow \tau_2 \mid \langle \tau_1, \cdots, \tau_n \rangle \mid \exists t.\tau \\
\text{(terms)}\ e ::= & x \mid \mathtt{c} \mid \mathtt{code}\ (x{:}\tau).e \mid \mathtt{call}\ e_1(e_2) \mid \\
& \langle e_1, \cdots, e_n \rangle \mid \pi_i\, e \mid \\
& \mathtt{pack}[\tau', e]\ \mathtt{as}\ \tau \mid \mathtt{let}\ t, x = \mathtt{unpack}\ e'\ \mathtt{in}\ e \mid \\
& \mathtt{let}\ x = e'\ \mathtt{in}\ e
\end{array}$$

Products are introduced and eliminated in the usual fashion. Code types are introduced by `code` terms and eliminated by `call` terms. Existentials are introduced by `pack` terms and eliminated by `unpack` terms. Terms also include a `let` construct which we use to simplify the translation.

The static semantics of $\lambda^{\exists}$ is also standard except for the `code` rule which is similar to the rule for λ-expressions, but requires that the code definition contain no free type variables or value variables. In other words, code definitions are always closed, and can thus always be defined at the top level.

The closure conversion translation is specified by giving a type translation, $\mathcal{T}[\![\ ]\!]$, mapping source types to target types, and a term translate mapping derivable source language typing judgments to target language terms. The type translation is defined as:

$$\begin{aligned}
\mathcal{T}[\![b]\!] &= b \\
\mathcal{T}[\![\tau_1 \to \tau_2]\!] &= \exists t.\langle (\langle t, \mathcal{T}[\![\tau_1]\!] \rangle \Rightarrow \mathcal{T}[\![\tau_2]\!]), t \rangle
\end{aligned}$$

The arrow type $\tau_1 \to \tau_2$ is translated to an existentially quantified value. Conceptually, the value is a pair of an abstract type (t) and a product value whose type depends upon t. In this situation, t will abstract the type of the environment of the given closure, thereby avoiding the typing problems mentioned in the introduction. The first component of the product value is code that takes a value of the abstract environment type and an argument of type $\mathcal{T}[\![\tau_1]\!]$, and yields a value of type $\mathcal{T}[\![\tau_2]\!]$. The second component of the product value is a value of the abstract environment type.

The term translation is given in Figure 1. The **var** and **b-I** cases are straightforward. For the **$\to$-I** case, we first translate the body of the λ-expression, extending Γ with $\{x{:}\tau_1\}$, to yield e'. We then create a closed piece of code that abstracts the free variables of e'. The code has a parameter x_{arg} which is always a pair consisting of the environment and the argument to the function. Upon invocation, the code extracts these parameters and binds them to x_{env} and x respectively. (We assume that x_{arg} and x_{env} are chosen so as to be distinct from all of the variables in the context.) The values for the variables occurring in Γ' are obtained by performing suitable projections on x_{env} and by binding the result to the appropriate variable via `let`. We

$$(\mathbf{var})\ \ \Gamma \uplus \{x{:}\tau\} \vdash_s x : \tau \rightsquigarrow x \qquad\qquad (\mathsf{b}\text{-}\mathbf{I})\ \ \Gamma \vdash_s \mathtt{c} : \mathsf{b} \rightsquigarrow \mathtt{c}$$

$$(\rightarrow\text{-}\mathbf{I})\ \ \frac{\Gamma \uplus \{x{:}\tau\} \vdash_s e : \tau' \rightsquigarrow e'}{\Gamma \vdash_s \lambda x{:}\tau.e : \tau \rightarrow \tau' \rightsquigarrow \mathtt{pack}[\tau_\Gamma, \langle v_{\mathrm{code}}, v_{\mathrm{env}}\rangle]\ \ \mathtt{as}\ \ \mathcal{T}[\![\tau \rightarrow \tau']\!]}$$

$$\begin{array}{lll}
\text{where } \Gamma & = & \{x_1{:}\tau_1, \cdots, x_n{:}\tau_n\} \\
\tau_\Gamma & = & \langle \mathcal{T}[\![\tau_1]\!], \cdots, \mathcal{T}[\![\tau_n]\!]\rangle \\
v_{\mathrm{env}} & = & \langle x_1, \cdots, x_n\rangle \\
v_{\mathrm{code}} & = & \mathtt{code}\ \ (x_{\mathrm{arg}}{:}\langle \tau_\Gamma, \mathcal{T}[\![\tau]\!]\rangle). \\
 & & \quad \mathtt{let}\ \ x_{\mathrm{env}} = \pi_1\, x_{\mathrm{arg}}\ \ \mathtt{in} \\
 & & \quad \mathtt{let}\ \ x = \pi_2\, x_{\mathrm{arg}}\ \ \mathtt{in} \\
 & & \quad \mathtt{let}\ \ x_1 = \pi_1\, x_{\mathrm{env}}\ \ \mathtt{in} \\
 & & \quad \vdots \\
 & & \quad \mathtt{let}\ \ x_n = \pi_n\, x_{\mathrm{env}}\ \ \mathtt{in} \\
 & & \quad e'
\end{array}$$

$$(\rightarrow\text{-}\mathbf{E})\ \ \frac{\Gamma \vdash_s e_1 : \tau_2 \rightarrow \tau \rightsquigarrow e_1' \qquad \Gamma \vdash_s e_2 : \tau_2 \rightsquigarrow e_2'}{\Gamma \vdash_s e_1\, e_2 : \tau \rightsquigarrow} \quad (x \notin Dom(\Gamma))$$
$$\mathtt{let}\ \ t, x = \mathtt{unpack}\ \ e_1'\ \ \mathtt{in}\ \ \mathtt{call}\ (\pi_1\, x)(\langle \pi_2\, x, e_2'\rangle)$$

Fig. 1. Closure Conversion for $\lambda^{\rightarrow}$

create the environment by building a tuple $\langle x_1, \cdots, x_n\rangle$ containing the values of those variables in Γ. The code and the environment tuple are placed in a pair, and the data structure is packed in order to abstract the type of the environment.

For the $\rightarrow$-**E** case, we translate the function and argument to yield e_1' and e_2' respectively. We then unpack e_1' and bind the abstract type to the type variable t and the contents of the closure to x. We then project the code from x followed by the environment, and call the code passing it the environment and argument e_2'.

3 Recursive Closure Conversion

In this section, we show how to extend the simply-typed closure conversion to deal with recursive functions. Interestingly, there are many possible translations, each with different tradeoffs in terms of efficiency and/or complexity of the resulting type system and semantics for the target language. A fascinating aspect is that all of the translations presented here have a direct correspondence to type encodings used for various kinds of object-oriented languages that support a notion of "self".

Our source language is the same as $\lambda^{\rightarrow}$, but we extend values with fix-expressions on abstractions:

$$\text{(terms) } e ::= \cdots \mid \texttt{fix } x(x_1{:}\tau_1){:}\tau_2.e$$

Here, both x and x_1 are bound within e. Though our source language only supports single recursion, it is straightforward but tedious to extend the presentation in the following sections to mutual recursion.

3.1 The Fix-Code Translation

In this section, we use recursive *code* definitions in the closure conversion translation. The translation is entirely straightforward in that the type translation is the same as for simply-typed closure conversion and no recursive (*i.e.*, cyclic) data structures are required. Hence, the "fix-code" translation is suitable for use with a reference-counting garbage collector. Furthermore, the fix-code translation supports easy optimization of known functions as with the original translation. However, the implementation leads to unecessary duplication of closures. In essence, a copy of the closure is re-created each time the function is invoked.

The only change to the target language is the addition of a `fixcode` construct:

$$\text{(terms) } e ::= \cdots \mid \texttt{fixcode } x(x{:}\tau_1){:}\tau_2.e$$

The code may only refer to its arguments or itself. The dynamic semantics "unrolls" the code at the point where it is called, just as fix is normally unrolled.

Closure conversion from the source to the target is straightforward. The type translation remains the same, as does the translation of application and

λ. The only addition is the translation rule for `fix`:

$$\textbf{(fix-1)}\quad \frac{\Gamma \uplus \{x{:}\tau' \to \tau, x'{:}\tau'\} \vdash_s e : \tau \leadsto e'}{\Gamma \vdash_s \texttt{fix}\ x(x'{:}\tau'){:}\tau.e : \tau' \to \tau \leadsto \texttt{pack}[\tau_\Gamma, \langle v_{\mathrm{code}}, v_{\mathrm{env}}\rangle]\ \texttt{as}\ \mathcal{T}[\![\tau \to \tau']\!]}$$

$$\begin{array}{lll}
\text{where } \Gamma & = & \{x_1{:}\tau_1, \cdots, x_n{:}\tau_n\} \\
\tau_\Gamma & = & \langle \mathcal{T}[\![\tau_1]\!], \cdots, \mathcal{T}[\![\tau_n]\!]\rangle \\
v_{\mathrm{env}} & = & \langle x_1, \cdots, x_n\rangle \\
v_{\mathrm{code}} & = & \texttt{fixcode}\ x_c(x_{\mathrm{arg}}{:}\langle \tau_\Gamma, \mathcal{T}[\![\tau]\!]\rangle). \\
& & \quad \texttt{let}\ x_{\mathrm{env}} = \pi_1\, x_{\mathrm{arg}}\ \texttt{in} \\
& & \quad \texttt{let}\ x' = \pi_2\, x_{\mathrm{arg}}\ \texttt{in} \\
& & \quad \texttt{let}\ x = \texttt{pack}[\tau_\Gamma, \langle x_c, x_{\mathrm{env}}\rangle]\ \texttt{as}\ \mathcal{T}[\![\tau \to \tau']\!]\ \texttt{in} \\
& & \quad \texttt{let}\ x_1 = \pi_1\, x_{\mathrm{env}}\ \texttt{in} \\
& & \quad \vdots \\
& & \quad \texttt{let}\ x_n = \pi_n\, x_{\mathrm{env}}\ \texttt{in} \\
& & \quad e'
\end{array}$$

The environment consists of the free variables of the body of the function except for x (the function itself) and x' (the argument). The closure for x is constructed as before by pairing the code with the environment. However, e' is obtained from e in a context where x must be available as a closure – not just code, as the closure could "escape" (*i.e.*, be passed as an argument to another function.) Hence, once we enter the code, we have to construct a "new" copy of the closure for use within the body of the code. Therefore, we create a new closure from x_c and x_{env} and bind it to x. Of course, if x is only called within the body e, then the reductions can eliminate the unnecessary construction of this value. But in general, we have to allocate a new closure pair each time around the loop which in practice actually is quite costly. The other translations attempt to address this by constructing the closure exactly once.

Note that the situation is even worse for mutually recursive functions. Suppose we've defined $f_1, \cdots, f_n$ via a letrec. In general, we have to construct *each* of the closures for $f_1, \cdots, f_n$ out of the code and shared environment each time one of these functions is called.

3.2 The Fix-Pack Translation

In our second translation, the key idea is that instead of adding recursive code to the target language, we add a recursive `pack` construct that allows us to define a closure data structure in terms of itself. The well-known trick is to build a circular data structure to represent `fix` closures where the environment for the closure contains the closure itself. In a sense, the previous translation is building this circular data structure lazily each time the closure is invoked. Our goal with this translation is to build the circular data structure once, avoiding the overhead of allocating a new copy each time the code is invoked.

To accomplish this, we add to the original target language the following term constructs:

$$\text{(terms) } e ::= \cdots \mid \texttt{fixpack } x.[\tau', v] \texttt{ as } \tau$$

The new construct allows one to define a recursive package of existential type. Like a recursive function, the variable x is bound within the body v of the term.

With the new target language construct, we define the recursive closure conversion translation as follows: The type translation is the same as for the first two translations. The term translation is also the same for application and λ, but the translation for `fix` is:

$$\textbf{(fix-2)}\ \frac{\Gamma \uplus \{x{:}\tau \to \tau', x'{:}\tau\} \vdash_s e : \tau' \rightsquigarrow e'}{\Gamma \vdash_s \texttt{fix } x(x'{:}\tau){:}\tau'.e : \tau \to \tau' \rightsquigarrow}$$

$$\texttt{fixpack } x.[\tau_{\text{env}}, \langle v_{\text{code}}, v_{\text{env}}\rangle] \texttt{ as } \mathcal{T}[\![\tau \to \tau']\!]$$

$$\begin{aligned}
\text{where } \Gamma &= \{y_1{:}\tau_1, \cdots, y_n{:}\tau_n\} \\
\tau_{\text{env}} &= \langle \mathcal{T}[\![\tau \to \tau']\!], \mathcal{T}[\![\tau_1]\!], \cdots, \mathcal{T}[\![\tau_n]\!]\rangle \\
v_{\text{env}} &= \langle x, y_1, \cdots, y_n\rangle \\
v_{\text{code}} &= \texttt{code } (x_{\text{arg}} : \langle \tau_{\text{env}}, \mathcal{T}[\![\tau]\!]\rangle) : \mathcal{T}[\![\tau']\!]. \\
&\qquad \texttt{let } x_{\text{env}} = \pi_1\, x_{\text{arg}} \texttt{ in} \\
&\qquad \texttt{let } x' = \pi_2\, x_{\text{arg}} \texttt{ in} \\
&\qquad \texttt{let } x = \pi_1\, x_{\text{env}} \texttt{ in} \\
&\qquad \texttt{let } y_1 = \pi_2\, x_{\text{env}} \texttt{ in} \\
&\qquad \vdots \\
&\qquad \texttt{let } y_n = \pi_{n+1}\, x_{\text{env}} \texttt{ in} \\
&\qquad e'
\end{aligned}$$

3.3 The Fix-Type translation

The previous translations used an environment-passing strategy where the code of a closure is passed the environment as an extra argument. In this translation, instead of passing the environment, we pass the closure itself as an extra argument to the code of the closure. Right away, it's obvious that the closure must contain a recursive type because the code is contained in the closure, but the code takes the closure as an argument. Hence, a possible type translation is given by:

$$\mathcal{T}[\![b]\!] = b$$

$$\mathcal{T}[\![\tau_1 \to \tau_2]\!] = \exists t_{\text{env}}.\mu t_{\text{cl}}.\langle (t_{\text{cl}}, \mathcal{T}[\![\tau_1]\!]) \Rightarrow \mathcal{T}[\![\tau_2]\!], t_{\text{env}} \rangle$$

Given this, the previous translations for application are correct, modulo the insertion of an "unroll" (assuming we want the isomorphism between the rolled and unrolled versions of a recursive type to be made explicit):

$$(\textbf{app-3})\ \frac{\Gamma \vdash_s e_1 : \tau_2 \to \tau \leadsto e_1' \qquad \Gamma \vdash_s e_2 : \tau_2 \leadsto e_2'}{\begin{array}{c}\Gamma \vdash_s e_1\, e_2 : \tau \leadsto \\ \texttt{let}\ t, x = \texttt{unpack}\ e_1'\ \texttt{in}\ \texttt{call}\ (\pi_1\,(\texttt{unroll}(x)))(x, e_2')\end{array}}$$

The term translation for fix becomes:

$$(\textbf{fix-3})\ \frac{\Gamma \uplus \{x{:}\tau_1 \to \tau_2, x'{:}\tau_1\} \vdash_s e : \tau_2 \leadsto e'}{\begin{array}{c}\Gamma \vdash_s \texttt{fix}\ x(x'{:}\tau_1){:}\tau_2.e : \tau_1 \to \tau_2 \leadsto \\ \texttt{pack}[\tau_\Gamma, \texttt{roll}(\langle v_{\text{code}}, v_{\text{env}}\rangle{:}\tau_{\text{cl}})]\ \texttt{as}\ \mathcal{T}[\![\tau_1 \to \tau_2]\!]\end{array}}$$

$$\begin{array}{lll}
\text{where}\ \Gamma & = & \{x_1{:}\tau_1, \cdots, x_n{:}\tau_n\} \\
\tau_\Gamma & = & \langle \mathcal{T}[\![\tau_1]\!], \cdots, \mathcal{T}[\![\tau_n]\!] \rangle \\
\tau_{\text{cl}} & = & \mu t_{\text{cl}}.\langle (t_{\text{cl}}, \mathcal{T}[\![\tau_1]\!]) \Rightarrow \mathcal{T}[\![\tau_2]\!], \tau_\Gamma \rangle \\
v_{\text{env}} & = & \langle x_1, \cdots, x_n \rangle \\
v_{\text{code}} & = & \texttt{code}\ x_c(x_{\text{cl}}{:}\tau_{\text{cl}}, x{:}\mathcal{T}[\![\tau_1]\!]) : \mathcal{T}[\![\tau_2]\!]. \\
 & & \quad \texttt{let}\ x = \texttt{pack}[\tau_\Gamma, x_{\text{cl}}]\ \texttt{as}\ \mathcal{T}[\![\tau_1 \to \tau_2]\!]\ \texttt{in} \\
 & & \quad \texttt{let}\ x_{\text{env}} = \pi_2\,(\texttt{unroll}(x_{\text{cl}}))\ \texttt{in} \\
 & & \quad \texttt{let}\ x_1 = \pi_1\, x_{\text{env}}\ \texttt{in} \\
 & & \quad \vdots \\
 & & \quad \texttt{let}\ x_n = \pi_n\, x_{\text{env}}\ \texttt{in} \\
 & & \quad e'
\end{array}$$

This translation has all of the (operational) advantages of the fixpack approach. In particular, the closure is only created once and not each time around the loop as in the fixcode case.

There are a couple of relatively minor reasons why this translation might be preferred over the fixpack translation: In particular, since the code, not the caller, extracts the environment from the closure, a slight optimization is possible when the environment is empty, as the code can avoid projecting the environment. In contrast, for all of the other translations, the caller extracts the environment. Since the caller cannot in general know the type of the environment (*i.e.*, whether it is unit), the caller cannot know whether the environment is actually needed or not. Furthermore, with suitable support in the target language, the environment can be "flattened" into the closure tuple. That is, instead of $\langle v_{\text{code}}, \langle v_1, \cdots, v_n \rangle\rangle$, we can represent the closure as $\langle v_{\text{code}}, v_1, \cdots, v_n \rangle$, thereby avoiding extra allocation and indirection. It is worth remarking that this is the closure-passing style that Appel and Jim proposed in an untyped setting [2], and was used until recently in the SML/NJ compiler [3].

Closure-passing does have its drawbacks: it requires recursive types (not just monotonic types) which seriously complicates the semantics of the target language, and to reap the allocation benefits, requires some rather *ad hoc* data structures (*e.g.*, pointers into the middle of tuples [1].)

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 *Sixteenth ACM Symposium on Principles of Programming Languages*, pages 293–302, Austin, TX, January 1989.

[3] A. W. Appel and D. B. MacQueen. Standard ML of New Jersey. In J. Maluszynski and M. Wirsing, editors, *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13, New York, August 1991. Springer-Verlag.

[4] L. Birkedal and R. Harper. Relational interpretations of recursive types in an operational setting (summary). In *Theoretical Aspects of Computer Science*, Sendai, Japan, September 1997. (To appear.).

[5] I. A. Mason, S. F. Smith, and C. L. Talcott. From operational semantics to domain theory. *Information and Computation*, 128(1):26–47, 1996.

[6] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *Twenty-Third ACM Symposium on Principles of Programming Languages*, pages 271–283, St. Petersburg, FL, January 1996.

[7] J. C. Mitchell and G. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988.

[8] B. C. Pierce and D. N. Turner. Object-oriented programming without recursive types. In *Twentieth ACM Symposium on Principles of Programming Languages*, Charleston, SC, Jan. 1993.

[9] A. M. Pitts. Relational properties of domains. *Information and Computation*, 127(2):66–90, June 1996.