A Comprehensive Treatment of Typed Closure Conversion
ABSTRACT

Greg Morrisett and Robert Harper

November 3, 1997

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 a closed piece of *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 CPS conversion reifies meta-level continuations as object-level functions.

As an example, the source level expression:

$$
\begin{array}{lll}
\texttt{let} & f & = \lambda x.\lambda y.\lambda z.x+y+z \\
 & f' & = f\,3 \\
 & f'' & = f'\,4 \\
 & i & = f''\,5 \\
\texttt{in} & & \\
 & i & \\
\texttt{end} & &
\end{array}
$$

might be closure-converted to the target language expression:

$$
\begin{array}{lll}
\texttt{let} & z_{\rm code} & = \lambda[e,z].(\pi_2\,e)+(\pi_1\,e)+z \\
 & y_{\rm code} & = \lambda[e,y].\langle z_{\rm code}, \langle y, \pi_1\,e, \rangle\rangle \\
 & x_{\rm code} & = \lambda[e,x].\langle y_{\rm code}, \langle x\rangle\rangle \\
 & f & = \langle x_{\rm code}, \langle\rangle\rangle \\
 & f' & = (\pi_1\,f)\,[\pi_2\,f, 3] \\
 & f'' & = (\pi_1\,f')\,[\pi_2\,f', 4] \\
 & i & = (\pi_1\,f'')\,[\pi_2\,f'', 5] \\
\texttt{in} & & \\
 & i & \\
\texttt{end} & &
\end{array}
$$

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, 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 $\langle y_{\text{code}}, \langle 3\rangle\rangle$. 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 $\langle z_{\text{code}}, \langle 4, 3\rangle\rangle$. The definition of i 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 Minamide, Morrisett, and Harper [7], we considered the issue of type-preserving closure conversion. We noted that a naive treatment of closure conversion does not 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 $b{:}\mathsf{int}$, the expression may be assigned the type $\mathsf{int} \to \mathsf{int}$. A typical closure conversion algorithm yields the following target expression:

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

But the resulting term is not well-typed, because the closure in the **then** clause has type $([\mathsf{int} \times \mathsf{int}, \mathsf{int}] \to \mathsf{int}) \times \mathsf{int} \times \mathsf{int}$ whereas the closure in the `else` clause has type $([\langle\,\rangle, \mathsf{int}] \to \mathsf{int}) \times \langle\,\rangle$. 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.

We then showed that a type-preserving closure conversion for the simply-typed λ-calculus was possible through the use of existential types. The critial idea is to abstract the type of the environment when forming the closure. Hence, source functions of type $\tau_1 \to \tau_2$ are translated to target closures with type $\exists\alpha.([\alpha, \tau_1] \to \tau_2) \times \alpha$. In essence, we translate functions to Pierce-Turner objects [9], consisting of one method (the code) and one instance variable (the environment).

We also considered a type-preserving closure conversion for polymorphic programming languages, where types are constructed and passed to polymorphic routines at run-time in order to support implementation techniques such as tag-free garbage collection [11, 8] or intensional polymorphism [5]. This setting is considerably more complicated because the code must abstract both the free *type* variables, as well as the free value variables of a function. Furthermore, we must use abstract *kinds* as well as abstract types, so that we can hide both the kind of the type environment and the type of the value environment. Finally, the translation requires some mechanism to connect the code of a closure to its type and value environments in such a way as to reflect the proper instantiation of type variables in a particular context. This last point is a remarkably subtle aspect of polymorphic closure conversion, and while we did find an approach based on *translucent* types [4] that works, we were unhappy with our explanation as to why the translation worked, and why translucency seemed necessary.

We did not consider closure conversion for a language with recursion functions. One reason was that we discovered many different ways to encode recursion and at the time, no translation seemed to dominate the others. Another reason we did not consider recursive functions is that the techniques we used to establish correctness of the translations for the the non-recursive source languages were not easily extended to cover recursion. In particular, we used Tait's method of logical relations to prove the correctness of the simply-typed and predicative polymorphic closure conversion algorithms, and Girard's method of candidates to prove the correctness of closure conversion for an impredicative polymorphic source language. Some of our translations relied upon reflexive types (i.e., recursive types with negative occurrences of the recursively-bound type variable). Extending Tait's method to deal with reflexive types seems tractible in light of recent work by Pitts [10] and Harper and Birkedal [2], though by no means straightforward. Extending Girard's method of candidates also seems possible, but technically daunting.

We have recently extended our previous work by:

1. giving a simpler framework for proving the correctness of closure conversion. The framework is powerful enough to prove the correctness of closure conversion for impredicative, polymorphic languages with recursive functions. Furthermore, the arguments of correctness are relatively simple.

2. treating closure conversion of recursive functions. We show that there are a variety of possible translations and, as in the simply-typed case, there is a strong connection between the these approaches and object encodings proposed by Abadi, Bruce, Cardelli, Pierce, Turner, and Viswanathan [9, 3, 1].

3. giving a simplified account of closure conversion for a *type-erasure* interpretation of polymorphism. In a type-erasure setting, types are erased before execution. Many implementations of typed, functional languages assume such an approach, and indeed, the original *Definition of Standard ML* [6] assumed such an approach. The closure conversion algorithm for this case is much simpler than for the type-passing case in that it requires neither abstract kinds nor translucent types. The drawback is that the approach cannot support techniques such as tag-free garbage collection or intensional polymorphism.

4. giving a more thourough explanation of polymorphic closure conversion for a type-passing interpretation of polymorphism. In particular, we show that translucency is a critical component for preserving type-sharing in compilers that implement ML-style functors through the phase-splitting transformation proposed by Harper, Moggi, and Mitchell.