Combining Closure Conversion with Closure Analysis using Algebraic Types

Andrew Tolmach*
Pacific Software Research Center
Portland State University & Oregon Graduate Institute
apt@cs.pdx.edu

May 2, 1997

Abstract

We have developed a new approach to typed closure conversion of higher-order monomorphic languages which also provides a useful uniform framework for closure analysis and subsequent optimization. The approach is based on representing closures as ordinary algebraic data constructors, grouped into algebraic datatypes according to source type; no special type system is needed. Closure applications are interpreted in the style of Reynolds [13]. Programs in our closure-converted form can also be viewed as representing the results of a closure analysis, which can be used to support many optimization strategies including partial evaluation, inlining, and stack-allocation of closures. The closure conversion itself provides a "free" level of closure analysis, based on type distinctions in the user program; to our knowledge, no previous system has performed such a type-based closure analysis. In addition, the results of more precise automated closure analyses can also be recorded in our form; we describe one such analysis based on unification. Our approach has been implemented for monomorphic, entire programs; we also sketch extensions for separate compilation.

1 Introduction

Many modern high-level languages support higher-order functions. Since most target languages, such as assembly code and C ("the universal portable assembler"), don't support first-class treatment of nested functions, compilers must transform higher-order programs into first-order ones, i.e., perform closure conversion. This transformation is typically performed on a λ-calculus-like intermediate language (IL); for simplicity, and to permit subsequent optimization, it is desirable to express the first-order result in a subset of the same IL, as in "closure-passing style" [2]. Moreover, there are considerable benefits to using a *typed* IL for both source and target of the conversion; some

*Supported, in part, by the US Air Force Materiel Command under contract F19628-93-C-0069 and by the National Science Foundation under grant CCR-9503383.

compiler targets require typed code,[1] and typed ILs are extremely useful for finding compiler bugs. Typing closure records is challenging because the type of the closed-over function does not reflect the types of the free variables that appear in the closure. Minamide, Morrisett, and Harper [10] have recently treated this problem. Their solutions rely on introducing new language primitives for closure manipulation or on introducing existential types; both approaches complicate subsequent optimization, and neither leads to typable code in a target language such as C.

Higher-order functions also complicate compilers by making flow analysis much more difficult: data flow and control flow become interdependent, so analyses from the conventional 3GL compiler world won't work without modification. Many partial-evaluation-based optimizations, such as value propagation and dead-code elimination, require the compiler to determine an (approximation of) the set of λ-expressions that might be invoked at each application site in the IL program. Existing implementations of this so-called *closure analysis* use an abstract interpretation involving a fixpoint calculation [14, 15] or a constraint-based mechanism [6, 12]. Surprisingly, closure analysis does not appear to have been implemented for typed languages, despite the fact that typing obviously provides a good first cut at the analysis "for free." Also, existing closure analysis algorithms do not express their results within IL itself, and so cannot feed subsequent general-purpose optimizations.

We have developed a new approach to typed closure conversion and typed closure analysis for monomorphic higher-order languages, based on the following observations. Closure conversion ordinarily produces a record for each closure value, containing a pointer to the code of the lambda-expression that produced the closure, together with values of any free variables;[2] closures are invoked by making an indirect jump through the code pointer, passing the free variable record as extra arguments. But inspired by Reynolds [13] or Warren [16], we can replace the code pointers with tags that uniquely identify the lambda-expressions, together with a dispatch function that inspects the tag and invokes the correct corresponding lambda; closures are now invoked by calling the dispatch function with the relevant tag and free variable record. To maintain well-typedness, each distinct arrow type in the source program must give rise to a distinct dispatch function and associated set of tagged records [3, 5, 4]. But sets of tagged records are simply *algebraic data types* (discriminated unions), such as are already present in most functional languages! Moreover, the set of tags for the dispatch function invoked for a given closure application site represents a (conservative) closure analysis indicating what functions can be invoked at that site. In short, the results of closure conversion and closure analysis can both be represented in an explicitly typed IL containing algebraic data types. Moreover, many standard, general-purpose optimizations for such an IL will perform useful improvements on closure-converted code.

We take a simple approach to closure conversion; it applies to explicitly typed, monomorphic ILs, and works best when the entire program is available for analysis and transformation (but see Section 5). The approach shares much with the work of Bell, Bellegarde and Hook [4], who present

[1] Our work in this area was originally motivated by the need to translate *into* strongly-typed high-level languages; the overall compiler context is described in [11].

[2] We assume a *flat* closure representation in this paper, but more elaborate representations can also be handled by our framework.

Original code:

```
fun outer (y:int,q:int list) =
  let fun g1 (x:int) : int = x + y
      fun g2 (x:int) : int = x - 2
      fun map (f:int->int,l:int list) : int list =
               case l of
                 Nil => Nil
               | Cons(h,t) => Cons(f h,map (f,t))
  in map (g1, q)
  end
```

After closure-converting g1 and g2:

```
datatype clos = G1 of int | G2          (* closure datatype *)
fun g1' (x:int,y:int) : int = x + y     (* lambda-lifted functions *)
and g2' (x:int) : int = x - 2
and dispatch(c:clos,i:int) : int =      (* dispatch function *)
      case c of
        F y => g1'(i,y)
      | G => g2'(i)

fun outer (y:int,q:int list) =
  let val g1'':clos = G1 y              (* closures *)
      val g2'':clos = G2
      fun map(f:clos,l:int list) : int list =
            case l of
              Nil => Nil
            | Cons(h,t) => Cons(dispatch(f,h),map(f,t))
  in map (g1'', q)
  end
```

Figure 1: Simple example of typed closure conversion

a considerably more elaborate conversion algorithm that copes with polymorphic source programs, specializing them on the fly as necessary. The key idea is to generate a distinct closure datatype corresponding to each distinct arrow type in the source program. Each source lambda expression gives rise to a distinct data constructor, which is added to the appropriate closure datatype based on the lambda expression's type. To convert a program to first order, the closure type declarations are introduced, lambda expressions are transformed into constructor applications, and calls to "unknown" (i.e. lambda-bound) functions are transformed into calls to an auxiliary function that dispatches on the constructor to invoke a lambda-lifted version of the correct original function. To maintain well-typedness, there must be a distinct dispatch function for each source arrow type. As usual, calls to "known" (i.e. let-bound) functions need not be converted in this way — they are simply changed to invoke the lambda-lifted version; if all calls to a function are known, the overhead of constructing a closure datatype value can be avoided altogether. Section 2 gives some simple examples; the Appendix gives details of the algorithm.

The closure-converted program embodies a closure analysis based on the original program's types; Section 3 describes some of the optimizations that can be performed on the basis of this analysis. Section 4 describes the implementation of a more refined analysis, essentially similar to the constraint-based analysis of Bondorf and Jørgensen [6], and shows how its results a can be recorded in the same IL. Section 5 sketches an extension of our mechanism to separate compilation, and Section 6 describes ongoing work.

2 Examples

Figure 1 gives a simple example of the conversion, presented using SML syntax. In this case, all the closure values in the program have type `int -> int`, so there is only one closure datatype.

In general, however, multiple sets of datatypes and dispatch functions are needed. For example, if we alter the above example to use a continuation-passing version of `map`, as in Figure 2, two sets are needed. Executing this example generates a (statically) unbounded number of distinct `int list -> int list` closures, one for each member of `q`, each containing another such closure as a free variable. This is reflected in the fact that type `clos_il2il` is recursive. Note that higher-order removal techniques based on code specialization [8] cannot cope with programs of this sort.

3 Utilizing Type-based Closure Analysis

After conversion and simple clean-up optimizations, a typical call to an unknown function has become a known call (to a dispatch function) followed by a case dispatch. Conventionally closure-converted programs would just do an indirect jump; happily, for most target languages, this more efficient form can be recovered as a final compilation step by using the lifted functions' code pointers as the tags on the corresponding closure type's constructors.[3]

In any case, there are many potential performance *advantages* to be obtained from the "interpreted" style of the converted program, by noting that it represents the results of a simple type-based closure analysis. The set of lambdas that can be invoked at a particular application site is explicit in the dispatch function called at that site. Standard dataflow analysis can be performed directly on the resulting program, which enables a host of partial-evaluation style optimizations such as constant propagation and dead code elimination. In addition, there are potential optimization payoffs if the number of data constructors for a particular closure type is small. A singleton set of constructors is ideal: the optimizer know precisely which function will be called, and can arrange to call it directly or perhaps inline it [9]; inlining is also possible (with some risk of code blow-up) for sets with just a few constructors. We have implemented a simple iterative optimizer for the IL shown, supporting dead variable elimination, various forms of constant propagation, and inlining of functions that are very small or are called only once.

If a closure datatype must be built, the compiler can use the fact that it knows all the constructors to choose an optimized representation. The standard datatype representation tricks [7, 1]

[3]But one of our compiler's target languages, Ada83, doesn't even support top-level function pointers. And conceivably some (future?) hardware processors will use speculative execution to produce better performance on case dispatch than on indirect jumps.

Original code:

```
fun outer (y:int, q:int list) =
  let fun g1 (x:int) : int = x + y
      fun g2 (x:int) : int = x - 2
      fun id (p:int list) : int list = p
      fun map (f:int->int,l:int list,k:int list -> int list) : int list =
               case l of
                 Nil => k(Nil)
               | Cons(h,t) =>
                        let fun kk(p:int list) : int list =
                                k(Cons(f h, p))
                        in map(f,t,kk)
                        end
  in map (g1,q,id)
  end
```

After closure-converting g1,g2,id, and kk:

```
datatype clos_i2i = G1 of int | G2                            (* int -> int closure datatype *)
and clos_il2il = ID | KK of clos_il2il * clos_i2i * int       (* int list -> int list closure datatype *)
fun g1' (x:int,y:int) : int = x + y                           (* lambda-lifted functions *)
and g2' (x:int) : int = x - 2
and id' (p:int list) : int list = p
and kk' (p:int list,k:clos_il2il,f:clos_i2i,h:int) : int list = (* k,f,h are free vars of kk *)
        dispatch_il2il(k, Cons(dispatch_i2i(f,h),p))

and dispatch_i2i(c:clos_i2i,i:int) : int =                    (* int->int dispatch function *)
      case c of
        G1 y => g1'(i,y)
      | G2 => g2'(i)
and dispatch_il2il(c:clos_il2il,p:int list) : int list =      (* int list -> int list dispatch function *)
      case c of
        ID => id'(p)
      | KK(k,f,h) => kk'(p,k,h,h)

fun outer (y:int,q:int list) =
  let val g1'':clos_i2i = G1 y                                (* closures *)
      val g2'':clos_i2i = G2
      val id'': clos_il2il = ID
      fun map(f:clos_i2i,l:int list,k:clos_il2il) : int list =
            case l of
              Nil => Nil
            | Cons(h,t) =>
                   let val kk'':clos_il2il = KK(k,f,h)
                   in map(f,t,kk'')
                        end
  in map (g1'', q, id'')
  end
```

Figure 2: Closure conversion at multiple types.

will avoid building heap records for closure constructors with no free variables. It is also useful to support "flat" (i.e., unboxed) variant types to avoid heap allocation for non-recursive constructors that have just a few free variables, such as `clos_i2i` in Figure 2.

The payoff from these optimizations depends on the precision of the underlying type-based closure analysis, and this in turn depends on source program types. To the extent that these types represent structural distinctions among values, they are essentially fixed by the shape of the program. However, source languages that support a name-equivalence model for types allow programmers to distinguish between different uses of structurally equivalent types. In ML, for example, this can be done by using "transparent" `datatype` declarations, e.g.,

```
datatype farenheit = F of integer
datatype centigrade = C of integer
```

Ordinarily, programmers use transparent declarations to make their program text clearer and to obtain help from the compiler's typechecker in detecting logical errors. For example, lambda-bound functions of type `farenheit -> farenheit` can be reliably distinguished from those of type `centigrade -> centigrade`, etc., reducing the risk of accidentally confusing the two kinds of quantities. In our scheme, these two functions will go into distinct closure datatypes, each having fewer constructors than would a datatype for their common structural type `int -> int`, and hence possibly offering more optimization opportunities at their call sites. Thus users have a further motive for making fine typing distinctions: they may thereby enable better optimization, more efficient closure representations, and better performance!

4 Recording Enhanced Closure Analysis

The compiler can also perform its own forms of flow analysis, and record the results in the form of a more specialized typing. The closure converter will take the more specialized typing into account when collecting constructors into closure datatypes, and produce a larger number of datatypes each containing fewer constructors.

We have built one such analyzer, based on ideas from type inferencing.[4] Beginning with the original program, in which every expression is annotated with an explicit (monomorphic) type, the analyzer tags each occurrence of an arrow type with a unique integer. It then performs a standard type-checking traversal of the program, with one adjustment: whenever the type-checker unifies two arrow types, the integer tags on these types are placed in the same equivalence class. In particular, this guarantees that if a lambda expression $(l : \tau_1 \rightarrow^i \tau_2)$ is among the lambdas that might possibly be applied at an application $(a : \tau_1 \rightarrow^j \tau_2)(b : \tau_1)$, then the tags i and j are necessarily in the same equivalence class. On the other hand, arrow tags are *not* placed in the same equivalence class merely because their argument and result types match. Thus the classes are refinement on ordinary types. This analysis is simple and almost linear (its complexity is limited by that of the union-find algorithm), given that we already have the typed intermediate form in hand. It appears

[4] Dino Oliva conceived and implemented this analyzer.

to produce essentially the same analysis as the constraint-based approach described by Bondorf and Jørgensen [6] and further analyzed by Palsberg [12].

An important point about our framework is that the *result* of an automated analysis like this can be expressed directly in our IL, and used as the basis of a (finer-grained) closure conversion. This is done by rewriting the IL program. For each equivalence class $\tau_1 \rightarrow^i \tau_2$, the analyzer simply invents a new unary datatype $\mathtt{D}^i$ = $\mathtt{C}^i$ of τ_2 and replaces all instances of $\tau_1 \rightarrow^i \tau_2$ by $\tau_1 \rightarrow D^i$, adding the necessary coercions to the program. These amount to a $\mathtt{C}^i$ construction around the body of each function of this type and a `case` on the result of each application of such a function. The resulting program is fed directly to the ordinary closure converter.

The coercions just mentioned carry no runtime cost (since the $\mathtt{D}^i$ are "transparent" constructors), but they do inhibit some further IL optimizations. We have therefore developed a clean-up transformation, to be applied *after* closure conversion, that gets rid of all transparent datatypes. In fact, it may prove useful to get rid of all non-structural typing distinctions at this point, rewriting the program to use a set of structurally-distinct canonical datatypes.

5 Extensions for Separate Compilation

We can extend our closure-conversion approach to support separate compilation using some very mild extensions to the type system. Consider first a naive approach. To keep the presentation simpler, assume that we're dealing with a single arrow type $\tau_1 \rightarrow \tau_2$.

Each compilation unit U^i is independently transformed essentially as before, generating a closure datatype D^i with an associated set of constructors $C_1^i, \ldots, C_{n_i}^i$, and dispatch function $\mathtt{dispatch}_{\mathtt{D}^i}$. However, all closure values are further wrapped to be members of a *global* datatype

$$\mathtt{datatype\ D = G_1\ of\ D^1\ \mid\ \ldots\ \mid\ G_n\ of\ D^n}$$

that uniquely identifies the compilation unit, and unknown function applications are transformed into calls to a *global* dispatch function $\mathtt{dispatch}_{\mathtt{D}}$. Here D is handled as an *extensible* datatype (much like exceptions in Standard ML), and the individual units compile calls to the dispatch function like any other external call.

At link-time, after all the individual units have been translated and compiled, the global dispatch function is defined thus:

```
fun dispatch_D (d:D,x_1:τ_1) : τ_2 =
      case d of
        G_1 (d^i) => dispatch_{D^i} (d^i,x_1)
      | ...
      | G_n (d^n) => dispatch_{D^n} (d^i,x_1)
```

This scheme could be improved operationally by choosing the various C_j^i to be all distinct and building a monolithic global dispatch function at link time to avoid the second level of dispatching indirection. But we cannot so easily get around the problem that the full set of datatype constructors is *not* known when the individual units are compiled, so we cannot take advantage of such knowledge for optimization or to choose a more efficient, non-uniform representation for D.

To regain this advantage, at least for *local* closure values, i.e., those both defined and applied in the same compilation unit, we can adopt a somewhat more complicated two-level scheme, as follows. We build both per-unit and global datatypes and dispatch functions as before, but we revert to using the per-unit constructors C^i to replace function values and invocations of the per-unit dispatch functions $dispatch_{D^i}$ to replace unknown function applications. We also add to each per-unit datatype D^i a special new constructor E^i `of` D, with corresponding dispatch case allowing an "escape" to the global dispatch function:

```
fun dispatch_{D^i}(d^i:D^i, x_1:τ_1) : τ_2 =
      case d^i of
          ...
        | E^i d => dispatch_D (d,x_1)
```

The advantage of this scheme is that local closure values can then use the per-unit datatype D^i, for which we can continue to choose a specialized, efficient representation[5] and per-unit `dispatch` functions. The disadvantage is that manipulating closure values non-locally is more onerous. A closure value v defined in unit U^i but applied in unit U^j must be coerced on export into the wrapped form $G^i(v)$ in the global datatype D, which is always given a uniform representation, and then coerced again on import to the doubly-wrapped form $E^j(G^i(v))$. Moeroever, to maintain separate compilability, the first coercion can be performed only in the exporting unit and the second only in the importing unit.

A closure value can be exported from U^i to U^j only by being passed as an argument to a U^j function or returned as a result from a U^i function.[6] For simplicity, consider just the case of a call $f(c)$, where f is defined in U^j and c is a closure for a function defined in U^i; the other cases are similar. Such a call might be direct or indirect. In the direct case, the application site knows that f is external, so it can do the necessary export wrapping; the import wrapping can be done at the entry point of f (this requires having distinguished entry points for externally visible functions, which is straightforward). In the indirect case, the export wrapping must be performed in the exporting unit's dispatch function, at the point when it discovers that the unknown function being invoked is itself external, and the import wrapping is done in a distinguished version of the import unit's dispatch function.

As with the simple scheme, it is not really necessary to have two levels of tags; what is essential is to coerce exported closure constructors into a uniform format that any other module can handle. Of course, there is some possibility that a function value will be wrapped for export, only to be reimported later and actually invoked within the same module in which it was originally defined. Since every local dispatch function can handle global constructors, even for closures that happen to be local, this doesn't cause any problems with correctness, but it will have suboptimal performance. There may be some scope for static removal of inverse coercions.

We have not implemented either of these schemes at present. The more complicated scheme

[5] Of course, the existence of the E^i constructor means we will never be able to use really powerful optimizations like those dependent on having a singleton set of constructors.

[6] We ignore the possibility of externally visible mutable variables.

will only be worthwhile if most unknown function applications invoke local closure values; we have no evidence on this point at present.

6 Ongoing Work

We have implemented most of the ideas in this paper (except for those of Section 5), but we have yet to do proper benchmarking to assess the benefit of closure-analysis-based optimizations. We are currently building a front-end for our IL to permit access to existing ML benchmark suites. Proper benchmarking is particularly important because some of the most appealing optimizations work disproportionately well on small programs.

We are also begun to apply conventional (FORTRAN-world) optimizers to our closure-converted code, in order to take advantage of well-developed dataflow frameworks that don't rely on inlining to propagate information.

Appendix: Conversion Algorithm Details

We give an informal description of the closure conversion algorithm for explicitly typed, monomorphic variant of Core ML, with true multi-argument functions and constructors. For simplicity, we assume that all functions are explicitly named.

The algorithm maintains a global dictionary mapping arrow types to unique closure datatypes. Closure datatypes $\mathtt{D}_\tau$ are created as a side-effect of translating arrow types; the data constructors belonging to each $\mathtt{D}_\tau$ are invented and recorded in a global data structure during the term translation. The algorithm also maintain a global set of lifted functions, grouped by (closure-converted) type.

Source types τ are translated to closure-converted types $|\tau|$ as follows: $|\tau| = \tau$ for primitive or user-defined types τ; $|\tau_1 \times \ldots \times \tau_n \to \tau_0| = \mathtt{D}_{|\tau_1| \times \ldots \times |\tau_n| \to |\tau_0|}$, the uniquely defined closure datatype corresponding to the arrow type named in its subscript. This datatype is created and added to the global dictionary when the corresponding arrow type is first encountered.

The core of the translation is a syntax-directed recursive traversal of source terms. Only function declarations, mentions, and type annotations change.

- Each source function declaration $f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau_0 = e$ is handled as follows:
  - The free variables of f, $FV(f) = \{v_1 : \mu_1, \ldots, v_m : \mu_m\}$ are computed.[7]
  - A lambda-lifted version of f is defined as $\overline{f}(x_1 : |\tau_1|, \ldots, x_n : |\tau_n|, v_1 : |\mu_1|, \ldots, v_m : |\mu_m|) : |\tau_0| = |e|$, where $|e|$ is the translation of e, and this is added to the global list of lifted functions.

[7] In fact, free variables are computed just once for each set of mutually recursive function declarations. The precise definition of "free" is slightly tricky: known functions called within e are *not* included among the free variables of f, but *their* free variables are[1].

- A fresh data constructor $C_f : |\mu_1| \times \ldots \times |\mu_m| \rightarrow \mathtt{D}_{|\tau_1|\times\ldots\times|\tau_n|\rightarrow|\tau_0|}$ is invented and added to the closure datatype definition.
- At the point of function declaration, a closure value declaration $c_f : \mathtt{D}_{|\tau_1|\times\ldots\times|\tau_n|\rightarrow|\tau_0|} = C_f(v_1, \ldots, v_m)$ is generated.

- Every top-level function declaration f is altered to invoke the corresponding lifted function, i.e., $f(x_1, \ldots, x_n) = \overline{f}(x_1, \ldots, x_n, v_1, \ldots, v_m)$ where $\{v_1, \ldots, v_m\} = FV(f)$. Nested function declarations are removed.

- Every known call $f(y_1, \ldots, y_n)$ is replaced by a call to the corresponding lambda-lifted function, $\overline{f}(y_1, \ldots, y_n, v_1, \ldots, v_m)$, where $\{v_1, \ldots, v_m\} = FV(f)$ (Note that any known call to f must be in the scope of the free variables of f.)

- Every other mention of function f is replaced by c_f.

- Every call $f(x_1, \ldots, x_n)$ to an unknown function of type $\tau_1 \times \ldots \times \tau_n \rightarrow \tau_0$ is replaced by a call to the relevant dispatch function, i.e., $\mathtt{dispatch}_{\mathtt{D}_{|\tau_1|\times\ldots\times|\tau_n|\rightarrow\tau_0}}(f, x_1, \ldots, x_n)$.

- Every type annotation τ is replaced by $|\tau|$.

When the entire program body has been translated, the full set of data constructors $\{C_{f_1}, \ldots, C_{f_p}\}$ for each closure datatype is known, and the corresponding dispatch functions can be built as follows:

$$
\begin{array}{l}
\mathtt{dispatch}_{\mathtt{D}_{|\tau_1|\times\ldots\times|\tau_n|\rightarrow\tau_0}}\ (c, x_1, \ldots, x_n)\ \mathtt{=} \\
\quad \mathtt{case}\ c\ \mathtt{of} \\
\qquad \mathtt{C}_{f_1}\ (v_1, \ldots, v_{m_1})\ \mathtt{=>}\ \overline{f}(x_1, \ldots, x_n, v_1, \ldots, v_{m_1}) \\
\quad\ \ \mathtt{|}\ \ldots \\
\quad\ \ \mathtt{|}\ \mathtt{C}_{f_p}\ (v_1, \ldots, v_{m_p})\ \mathtt{=>}\ \overline{f}(x_1, \ldots, x_n, v_1, \ldots, v_{m_p})
\end{array}
$$

The overall output of the translated program consists of the original user declarations, with translated types, and the new closure datatypes (all in a single mutually recursive set of declarations); the dispatch functions and lifted user functions (a single mutually recursive set of declarations); and the translations of the original top-level declarations.

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 Symp. on Principles of Programming Languages*, pages 293–302, New York, 1989. ACM Press.

[3] J. M. Bell. An implementation of Reynold's defunctionalization method for a modern functional language. Master's thesis, Oregon Graduate Institute, Jan. 1994.

[4] J. M. Bell, F. Bellegarde, and J. Hook. Type-driven defunctionalization. In *Proceedings Second International Conference on Functional Programming*, June 1997.

[5] F. Bellegarde and J. Hook. Substitution: A formal methods case study using monads and transformations. *Science of Computer Programming*, 23(2–3):287–311, 1994.

[6] A. Bondorf and J. Jørgensen. Efficient analyses for realistic off-line partial evaluation. *Journal of Functional Programming*, 3(3):315–346, 1993.

[7] L. Cardelli. Compiling a functional language. In *Proc. 1984 ACM Conference on Lisp and Functional Programming*, pages 208–217, Aug. 1984.

[8] W.-N. Chin and J. Darlington. Higher-order removal: A modular approach. Submitted for publication, 1994.

[9] S. Jagannathan and A. K. Wright. Flow-directed inlining. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 193–205, May 1996.

[10] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *Conference Record of POPL '96: 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 271–283, Jan. 1996.

[11] D. Oliva and A. Tolmach. From ML to Ada(!?!): Strongly-typed language interoperability via source translation. in preparation.

[12] J. Palsberg. Closure analysis in constraint form. *ACM Trans. Prog. Lang. Syst.*, 17(1):47–62, Jan. 1995.

[13] J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM National Conference*, pages 717–740. ACM, 1972.

[14] P. Sestoft. Replacing function parameters by global variables. Master's thesis, University of Copenhagen, Oct. 1988. DIKU Master's thesis no. 254.

[15] O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, May 1991. CMU-CS-91-145.

[16] D. Warren. Higher-order extensions to PROLOG: are they needed? In J. Hayes, D. Michie, and Y.-H. Pao, editors, *Machine Intelligence 10*, pages 441–454. Edinburgh University Press, 1982.