J. Functional Programming 1 (1): 1–000, January 1993

From ML to Ada: Strongly-typed Language Interoperability via Source Translation[1]

ANDREW TOLMACH
Pacific Software Research Center
Department of Computer Science, Portland State University,
P.O. Box 751, Portland, OR, 97207-0751, USA
email: apt@cs.pdx.edu

DINO P. OLIVA[2]
Pacific Software Research Center
Department of Computer Science and Engineering, Oregon Graduate Institute,
20000 N.W. Walker Road, P.O. Box 91000, Portland, OR 97291-1000, USA
email: oliva@cse.ogi.edu

Abstract

We describe a system that supports source-level integration of ML-like functional language code with ANSI C or Ada83 code. The system works by translating the functional code into type-correct, "vanilla" C or Ada; it offers simple, efficient, type-safe inter-operation between new functional code components and "legacy" third-generation-language components. Our translator represents a novel synthesis of techniques including user-parameterized specification of primitive types and operators; removal of polymorphism by code specialization; removal of higher-order functions using closure datatypes and interpretation; and aggressive optimization of the resulting first-order code, which can be viewed as encoding the result of a closure analysis. Programs remain fully typed at every stage of the translation process, using only simple, standard type systems. Target code runs at speeds comparable to the output of current optimizing ML compilers, even though handicapped by a conservative garbage collector.

1 Introduction

Functional languages (*FLs*) such as ML and Haskell provide powerful and high-level control mechanisms and symbolic data types that are not available in traditional "third-generation" languages (*3GLs*) such as C, Ada, or Modula. For example, it is easier to define and iterate over a list in ML than in C. These high-level features

[1] This work was supported, in part, by the US Air Force Materiel Command under contract F19628-93-C-0069. The first author was also supported, in part, by the National Science Foundation under grant CCR-9503383. A portion of this work was previously presented by the first author at the 1997 Workshop on Types in Compilation, under the title "Combining Closure Conversion with Closure Analysis using Algebraic Types."

[2] Current address: Bell Laboratories, Rm 2C-408, 600 Mountain Ave., Murray Hill NJ 07974-0636, USA.

make FLs well-suited for rapid prototyping and stand-alone applications. But many real-world applications need to take advantage of an existing base of "legacy" code written in imperative 3GLs. Thus a reasonable aim is to enable programmers to use an FL to write "glue" code that combines existing 3GL code components, or to write FL components that can be integrated into larger 3GL-based systems.

Unfortunately, FL implementations typically do not give the programmer control over the detailed layout and lifetime of data, and usually assume a special-purpose runtime system; these characteristics impede interfacing with foreign languages. "Foreign function" interfaces that address these problems are becoming more common (Huelsbergen, 1996; Leroy, 1997; Peyton Jones et al., 1997; Tofte et al., 1997), but tend to have several disadvantages: moving data between languages typically requires expensive on-the-fly format conversions or tricky cast operations; there is often substantial overhead in transferring control between FL and 3GL runtime systems, which discourages small-grained interactions; and the resulting integrated code is an inelegant hybrid that depends on the implementation details of FL and 3GL compilers, which may be unacceptable in organizations that mandate use of standardized, portable 3GLs.

We have developed an alternative approach to interoperability that completely bypasses these problems by *translating* the entire FL program into the imperative 3GL used by the legacy code base. Specifically, we have built a system that translates an ML-like source language (called *RML*, for "Restricted ML") into well-typed, portable, "vanilla" Ada83 or ANSI C code, which can be passed to a standard compiler. Since the output of the translator represents the FL's types and control structures using the 3GL's types and control structures, FL and 3GL code can be easily integrated, even within a single procedure, in an efficient and fully type-safe manner.

Our system has been developed as the back end of a larger application generator system that produces integrable components from high-level specifications (Kieburtz et al., 1995); we first generate RML code from the specifications using semantics-directed techniques, and then translate that code to Ada83 using the scheme described in this paper.[1] However, the system is quite general; it can accommodate hand-written or generated RML code from any source, and may be useful in any context where tight integration with an existing legacy code base is desirable.

This paper describes the design and implementation of our RML-to-3GL translator. Many of the requirements on such a translator are familiar from existing FL compilers: high-level features such as polymorphism, higher-order functions, and algebraic datatypes need to be expressed in terms of much lower-level type and control constructs. However, the need to generate adequately performing, well-typed, vanilla target code—particularly for Ada83, a quite secure and restrictive language—makes special demands on the translator. These have led us to a novel combination of compilation techniques, some of which are of independent interest:

[1] The choice of Ada83 was mandated by our project sponsor, the U.S. Air Force Materiel Command.

- We use a type-based macro-expansion technique called *templates* (Volpano and Kieburtz, 1985; Volpano and Kieburtz, 1989) to integrate 3GL code into RML. Each RML program is translated with respect to a particular template, which defines a set of abstract, primitive types and a set of primitive operators. The definitions take the form of macros that expand into target-language text. Both substantial legacy code components and simple primitive types and operators are handled uniformly in this fashion. Operators must be monomorphic and first-order. Templates are specified using a specialized definition language; see Section 4.
- Our system removes polymorphism from RML programs by *cloning* polymorphic functions and datatype declarations, making a separate monomorphic version for each distinct set of instantiating type variables; see Section 6. Although this approach has been suggested before (Jones, 1993), and similar effects have sometimes been achieved by accident (Tarditi et al., 1996), we are unaware of any previous practical, intentional realizations. The approach requires access to the whole RML program.
- Our system removes higher-order functions using a novel *typed closure-conversion* algorithm that represents closures as members of algebraic datatypes, and generates type-specific dispatch functions to interpret them; see Section 9. The resulting code does not even require function pointers (which Ada83 lacks). Unlike previous treatments of typed closure-conversion (Minamide et al., 1996), we do not need to introduce new language primitives or fancy type systems to maintain typability, although our method does again require access to the whole program, which must be monomorphic.
- Our system *optimizes* the closure-converted code, using simple, standard "partial-evaluation-style" transformations; although optimizing at this stage has been suggested before, we are not aware that anyone has actually done it, and it proves to be useful. For example, the standard *uncurrying* optimization is performed "for free" by the standard inlining optimization; see Section 10.1.
- Furthermore, the code produced by our *typed* closure-conversion algorithm can be viewed as being the result of the simple, implicit *closure analysis.* Our system takes advantage of this closure analyses to choose more efficient closure representations and perform more aggressive inlining than an untyped conversion could support; see Section 10.2. We also show how to express the results of the somewhat stronger closure analysis of Bondorf and Jørgensen (Bondorf and Jørgensen, 1993; Palsberg, 1995) within the standard algebraic type framework; see Section 10.3.
- Our system eliminates *tail calls*, even among mutually recursive functions, without introducing global labels (which both ANSI C and Ada83 lack). It uses local labels instead, merging mutually recursive functions into a single function with multiple entry points if necessary; see Section 11.

The architecture of our translator resembles that of other recent transformation-based FL compilers (Appel, 1992; Leroy, 1991; Peyton Jones, 1996; Tarditi et al., 1996). The translator, which is itself written in Standard ML, is structured as a series of relatively simple transformations, each preserving semantics *and* types; see Section 5. It uses a small set of intermediate languages, each of which is strongly typed and executable by an interpreter. There are type-checkers and self-test mechanisms built in at each intermediate language stage; these have been used heavily during development to find and correct bugs in the translator. Only the very last transformation step is dependent on the particular 3GL target language involved, so the translator is easily retargeted to new output languages. Our system relies on standard 3GL compilers to handle traditional low-level concerns like register allocation, instruction selection, and local optimization, with reasonable results. Although high performance is not our primary goal, the performance of the C code generated by the translator compares favorably with the output of the well-regarded Standard ML of New Jersey compiler.

Memory management is one area in which we have not innovated. Our C back end incorporates the Boehm-Demers-Weiser conservative collector (Boehm and Weiser, 1988). Heap allocation in C is the one spot where we must perform casting, in order to allocate storage for values of different types from a common heap; of course, this is standard practice in C. Although Ada83 supports garbage collection in principle, the implementations we are using do not; the Ada-based applications we have built so far are structured so that it is safe to perform simple "bulk" deallocation (in the Ada code) at a few key points.

There has been much recent interest in using typed intermediate representations in compilers (Peyton Jones et al., 1993; Morrisett, 1995), but in most cases types are abandoned well before code generation. The TIL compiler (Tarditi et al., 1996) does keep type information until a late stage in the compilation process when code has reached a low level form more primitive than 3GL code, but its type system is substantially more complex than the C or Ada-style typing we use. While there are many existing systems that compile ML or Haskell to C (Tarditi et al., 1992; Cridlig, 1992; Chailloux, 1992; Peyton Jones, 1992; Tofte et al., 1997), they often make heavy use of casts or non-standard extensions (e.g., as provided by `gcc`), especially to handle closures and exceptions and to avoid using the C procedure activation model. Our system generates ANSI-standard, nearly cast-free code. Also, many of these systems generate C from very low-level intermediate forms, e.g., stack machines, so that the target C program has a completely different structure than the source FL program. By contrast, our translator only perturbs the function-level structure of the source program when it needs to; a "C-like" RML program with no nested or higher-order functions and no inter-function tail calls will be translated to a very natural-looking C program with the same structure. Our system does not currently handle exceptions, however.

This paper describes the overall architecture of our system, and reports in detail on the more novel transformations. We assume the reader to be familiar with the syntax of functional languages such as ML, and to be able to read Ada and C code. We have tried to avoid formality except as demanded for the sake of precision.

```
PACKAGE GeoLib IS
  TYPE trans_array IS ARRAY (1..3, 1..3) of float;
  TYPE transform IS ACCESS trans_array;
  TYPE point IS RECORD x:float; y:float; END RECORD;

  ID:transform := ...;
  FUNCTION rotate (r:float) RETURN transform;
  FUNCTION translate (x,y:float) RETURN transform;
  ...
  FUNCTION compose (x,y:transform) RETURN transform;
  FUNCTION apply (t:transform; p:point) RETURN point;
END GeoLib;
```

Fig. 1. Example Ada package specification (excerpts).

2 Example

As a simple motivating example, suppose we wish to build an RML component using an existing Ada package that implements simple 2D transformations on points (see Figure 1). Points are represented as pairs of reals and transformations as heap-allocated 3x3 real matrices; transformations are composed and applied using matrix multiplication (see Figure 2).

We want to use this existing Ada to do the numerical computation, while using RML for convenient manipulation of points and transforms considered as abstract values.[2] (We will also use Ada to write the "main program" or *driver* that will be responsible for invoking the RML component; we have little more to say about this driver, however.) In this application the granularity of the primitive operations is quite small, so invoking a function to perform each one might be quite inefficient. A template definition that imports these operations (and basic real number support) into an RML component is shown in Figure 3. This template declares `real`, `point` and `transform` as new abstract types, with the operator signatures as listed. Most of the operators expand into calls to the corresponding Ada routines; `apply` is defined to expand into inline Ada code. Template syntax is explained in Section 4.

A simple RML component that uses this template is shown in Figure 4. RML concrete syntax is similar to SML; details are given in Section 3. This component makes heavy use of RML's facility for defining and manipulating polymorphic algebraic types like `list` and abstract traversal operations like `foldl`. It builds a `list` of `transforms` and uses `foldl` and `compose` to make a combined transformation; it then uses another `foldl` to apply the combined `transform` to a `list` of `points`, and a third `foldl` to reverse the result (returning the list of transformed points to its original order).

The remainder of the paper will refer repeatedly to this example component, to

[2] This is a somewhat artificial example, since many functional language implementations have good built-in support for numerical computing, and recoding such a small legacy component would be easy.

```
PACKAGE BODY GeoLib IS
  ...
  FUNCTION rotate (r:float) RETURN transform IS
  BEGIN
    RETURN NEW trans_array'((cos(r),-sin(r),0.0),
                            (sin(r), cos(r),0.0),
                            (   0.0,    0.0,1.0));
  END rotate;
  ...
  FUNCTION compose (x,y:transform) RETURN transform IS
    ret_val:transform;
  BEGIN
    FOR i IN 1..3 LOOP  FOR j IN 1..3 LOOP ...  END LOOP; END LOOP;
    RETURN (ret_val);
  END compose;

  FUNCTION apply (t:transform; p:point) RETURN point IS
    ret_val:point;
  BEGIN
    --  N.B. Bottom row of t is always (0.0,0.0,1.0)
    ret_val.x := (p.x * t(1,1)) + (p.y * t(1,2)) + t (1,3);
    ret_val.y := (p.x * t(2,1)) + (p.y * t(2,2)) + t (2,3);
    RETURN (ret_val);
  END apply;
END GeoLib;
```

Fig. 2. Example Ada package implementation (Excerpts)

```
template GeoLibTemplate

type real "float"
type point "point"
type transform "transform"

val add (x0:real,x1:real) : (res:real) pure "`res` := `x0` + `x1`;"
...
val div (x0:real,x1:real) : (res:real) "`res` := `x0` / `x1`;"
...
val id : transform "id"
val rotate (r:real) : (res:transform)  pure "`res` := rotate (`r`);"
...
val apply (t:transform,p:point) : (res:point) pure
     "BEGIN \
      \ `res`.x := ((`p`.x * `t`(1,1)) + (`p`.y * `t`(1,2)) + `t`(1,3)); \
      \ `res`.y := ((`p`.x * `t`(2,1)) + (`p`.y * `t`(2,2)) + `t`(2,3)); \
      \ END"
```

Fig. 3. Example template for geometric operations (excerpts).

```
export type point list "PList"
       val Nil  : point list "PNil"
       val Cons : point * point list -> point list "PCons"
       val doit : point list -> point list "doit"

datatype 'a list = Cons of 'a * 'a list | Nil

fun foldl (c,n,l) =
  case l of
    Nil => n
  | Cons(x,r) => foldl(c,c(x,n),r)

val ts = Cons(translate (2.0,~2.0),
              Cons(scale(1.0,0.5),
                   Cons(rotate((div(3.141592,2.0))),Nil)))

fun doit ps =
       let val whole_t = foldl (compose,id,ts)
           fun consapp (x,l) = Cons(apply(whole_t,x),l)
           val ps0 = foldl(consapp,Nil,ps)
       in foldl(Cons,Nil,ps0)
       end
```

Fig. 4. RML component using geometric template.

show the effect of various transformations. As a preview of the end product, we show the final output of the RML-to-Ada translator on this component in Figure 5. This is genuine output, except that we have renamed the variables and reformatted for better readability, and coalesced some variable declarations and initial assignments into declarations with initializers. The output code illustrates many of the key characteristics of our translation approach, although because of the extremely small size of the input program, the optimizer has done an unusually good job with it. The output is efficient first-order monomorphic code. The original polymorphic `foldl` function has been specialized into two monomorphic variants `foldl0` and `foldl1`, taking `transform` lists to `transforms` and `point` lists to `point` lists, respectively. The two possible functional arguments to `foldl1`, namely `Cons` and `consapp`, are represented as members of a discriminated record type `closure`. The discriminant tag indicates which function is required; the `consapp` variant, which carries the free variable `whole_t` as an associated value, must be dynamically constructed, whereas the `Cons` variant is statically defined. In either case the closure is small enough to be manipulated by value, rather than being heap-allocated. Moreover, since `Cons` and `consapp` are used *only* as arguments to `foldl`, their code is actually inlined into `foldl1`. The primitive Ada code for `apply`, used within `consapp`, has been inlined, as specified in the template. Even stronger optimization has been applied to `foldl0`: since `compose` is the *only* argument that can be passed to it, no closure is required at all, and its body is specialized to call the primitive Ada `compose` routine directly.

```
WITH GeoLib; USE GeoLib; WITH Math; USE Math;
PACKAGE Geo_package IS
  TYPE PList_item ;  TYPE PList IS ACCESS PList_item;
  TYPE PList_item IS RECORD PCons_0:point;  PCons_1:PList; END RECORD;
  FUNCTION PCons (PCons_0:point; PCons_1:PList) RETURN PList;
  PNil:PList := NULL;

  FUNCTION doit (ps:PList) RETURN PList;
END Geo_package;

PACKAGE BODY Geo_package IS
  TYPE TList_item ;  TYPE TList IS ACCESS TList_item;
  TYPE TList_item IS RECORD TCons_0:transform; TCons_1:TList; END RECORD;
  FUNCTION TCons (TCons_0:transform; TCons_1:TList) RETURN TList;
  TNil:TList := NULL;

  TYPE closure_constructors IS (cons_variant,consapp_variant);
  TYPE closure (constructor:closure_constructors := cons_variant) IS
  RECORD CASE constructor IS
    WHEN cons_variant => NULL;
    WHEN consapp_variant => whole_t:transform;
  END CASE; END RECORD;
  cons:closure(cons_variant);

  tf: float; t0:transform; t1:transform; t2:transform;
  vts0:TList; ts1:TList; ts:TList;

  FUNCTION PCons (p0:point; p1:PList) RETURN PList IS
  BEGIN return NEW PCons_item'(PCons_0 => p0, PCons_1 => p1); END;
  FUNCTION TCons (t0:transform; t1:TList) RETURN TList IS
  BEGIN return NEW TCons_item'(TCons_0 => t0, TCons_1 => t1); END;

  FUNCTION foldl0 (n:transform; l:TList) RETURN transform IS
    n0:transform := n; l0:TList := l;
  BEGIN
    GOTO JumpPoint0;
    <<JumpPoint0>>
    IF l0 = NULL THEN
      RETURN n0;
    ELSE
      DECLARE x : transform := l0.TCons_0; r: TList := l0.TCons_1;
              n : transform := compose(x,n0);
      BEGIN
        n0 := n; l0 := r;
        GOTO <<JumpPoint0>>;
      END;
    END IF;
  END foldl0;
```

Fig. 5. Generated Ada code corresponding to example (beginning).

```
  FUNCTION foldl1 (c:closure, n:PList; l:PList) RETURN PList IS
    c0 : closure := c; n0 : PList := n; l0 : PList := l;
  BEGIN
    goto JumpPoint1;
    <<JumpPoint1>>
    IF l0 = NULL THEN
      RETURN n0;
    ELSE
      DECLARE x : point := l0.PCons_0 ; r : PList := l0.PCons_1;
      BEGIN
        CASE c.constructor IS
          WHEN cons_variant =>
            DECLARE n: PList := PCons(x,n0);
            BEGIN
              c0 := c0; n0 := n; l0 := r;
              GOTO <<JumpPoint1>>;
            END;
          WHEN consapp_variant =>
            DECLARE whole_t : transform := c0.whole_t; p0 : point;
            BEGIN
              p0.x := ((x.x * whole_t(1,1)) +
                       (x.y * whole_t(1,2)) + whole_t(1,3));
              p0.y := ((x.x * whole_t(2,1)) +
                       (x.y * whole_t(2,2)) + whole_t(2,3));
              DECLARE n : PList := PCons(p0,n0);
              BEGIN
                c0 := c0; n0 := n; l0 := r;
                GOTO JumpPoint1;
              END;
            END;
        END CASE;
    END IF;
  END f1;

  FUNCTION doit (ps:PList) RETURN PList IS
    whole_t : transform :=  foldl0(id,ts);
    c : closure := (consapp_variant,whole_t);
    ps0 : PList := foldl1(c,PNil,ps);
    ps1 : PList := foldl1(cons,PNil,ps0);
  BEGIN
    RETURN ps1;
  END doit;

BEGIN
  t0 := translate(2.0,-2.0); t1 := scale(1.0,0.5);
  tf := 3.141592 / 2.0; t2 := rotate(tf);
  ts0 := TCons(t2,TNil); ts1 := TCons(t1,ts0); ts := TCons(t0,ts1);
END Geo_package;
```

Fig. 5 (cont.). Generated Ada code corresponding to example (conclusion).

$$
\begin{array}{rcll}
\tau & ::= & K & \text{(primitive types)} \\
& | & t & \text{(type variables)} \\
& | & (\langle\tau\rangle_{*}) \rightarrow \tau & \text{(function types)} \\
& | & (\langle\tau\rangle_{,})\, D & \text{(algebraic types)} \\[1ex]
\sigma & ::= & [\forall\langle t\rangle_{,}\,.]\tau & \text{(type schemes)} \\[1ex]
e & ::= & (k : K) & \text{(primitive constants)} \\
& | & (v : \tau) & \text{(variables)} \\
& | & e\,(\langle e\rangle_{,}) & \text{(function applications)} \\
& | & (c : \tau)\,(\langle e\rangle_{,}) & \text{(constructor applications)} \\
& | & p(\langle e\rangle_{,}) & \text{(primitive applications)} \\
& | & \texttt{fn}\ [\texttt{inline}]\ \mathit{rule} & \text{(anonymous abstractions)} \\
& | & \texttt{let}\ \mathit{vdecs}\ \texttt{in}\ e & \text{(local declarations)} \\
& | & \texttt{case}\ e\ \texttt{of}\ \langle (c : \tau)\ \mathit{rule}\rangle_{|} & \text{(destructuring)} \\[1ex]
\mathit{rule} & ::= & (\langle v : \tau\rangle_{,})\ \texttt{=>}\ e & \text{(rules)} \\[1ex]
\mathit{vdecs} & ::= & \texttt{val rec}\ \langle v : \sigma = \texttt{fn}\ [\texttt{inline}]\ \mathit{rule}\rangle_{\texttt{and}} & \text{(recursive function declarations)} \\
& | & \texttt{val}\ v : \sigma = e & \text{(value declarations)} \\[1ex]
\mathit{atdec} & ::= & (\langle t\rangle_{,})\, D[\texttt{flat}] = \left\langle c\ [\texttt{of}\ \langle\tau\rangle_{*}]\right\rangle_{|} & \text{(algebraic type declarations)} \\[1ex]
\mathit{atdecs} & ::= & \texttt{datatype}\ \langle \mathit{atdec}\rangle_{\texttt{and}} & \text{(mutually recursive declarations)} \\[1ex]
\mathit{export} & ::= & \texttt{type}\ \tau\ \texttt{"}\mathit{name}\texttt{"} & \text{(type exports)} \\
& | & \texttt{val}\ v : \tau\ \texttt{"}\mathit{name}\texttt{"} & \text{(value exports)} \\[1ex]
\mathit{comp} & ::= & \texttt{export}\ \langle \mathit{export}\rangle\ \langle \mathit{atdecs}\rangle\ \langle \mathit{vdecs}\rangle & \text{(components)}
\end{array}
$$

Fig. 6. RML Abstract Syntax. In this and other syntax descriptions, we use the notation $\langle x\rangle_{sep}$ to mean a sequence of zero or more x's separated by *sep*, and $[x]$ to mean an optional x. When giving examples written in the syntax, we generally omit the grouping parentheses `()` when no ambiguity results.

The only heap-allocated structures in the Ada program are the lists themselves, for which the translator has automatically chosen an efficient representation using one record per list item and the `NULL` pointer to represent the empty list; `point` lists use a completely flattened five-word record per item, with no indirection for the point pair or for the embedded reals. The tail-recursive calls in `foldl0` and `foldl1` have been converted to local jumps. The only major remaining optimizations to be performed by the Ada compiler are variable coalescing, jump-to-jump elimination, and loop invariant hoisting.

3 RML Source Language

RML is an eager language with first-class functions, algebraic datatypes and parametric (Hindley-Milner) polymorphism. Plain RML, without primitives, is essentially similar to the pure subset of core Standard ML (SML '97) (Milner et al., 1997), without nested patterns or many derived forms, but with the addition of true multi-argument functions and data constructors. Impure features such as references, arrays, and I/O can be added to the language via the template primitive mechanism (see Section 4), but exceptions are fundamentally missing. In this paper, we use a human-readable but still somewhat abstract syntax for RML (Figure 6) and the other intermediate languages used in the translator. In this representation, it is assumed that no identifier is bound twice. In practice, source code is fed to the RML translator using a more elaborate concrete syntax (very similar to SML syntax) with the usual lexical scoping rules, or, for machine-generated source, using an internal representation of the abstract syntax. The primary difference between concrete and abstract syntax is that the former is untyped; the system performs standard Hindley-Milner type inference (Hindley, 1969; Milner, 1978; Damas, 1984; Cardelli, 1987) to obtain the type-annotated abstract form. Also, the concrete syntax allows primitives and constructors to be used as first-class values whereas the abstract syntax permits them only in the operator position of applications; such first-class uses are automatically eta-expanded by the concrete syntax parser. Finally, the parser accepts and translates some of the common SML derived forms, e.g., `fun` for `val rec`.

RML's typing rules are largely standard, so we mention only distinctive points here. RML abstract syntax includes explicit type annotations on variable and constructor uses and type schemes on declarations. These annotations suffice to reconstruct the types of arbitrary terms. Different mentions of a `let`-bound (or top-level) function or of a constructor may, of course, have different types; for any given mention, the instantiating type expressions for the generic type variables can be determined by unifying the type annotation on the mention with the scheme annotation on the declaration. Like SML '97, RML adheres to the value restriction on polymorphic bindings (Wright, 1995), requires recursive bindings to be explicit function abstractions, and prohibits polymorphic recursion among functions.

Unlike SML '97, RML also prohibits polymorphic recursion in datatype definitions.[3] Also, unlike SML, RML lacks records or tuples *per se*, but these can be built as datatypes with a single constructor. Datatypes can be marked as "`flat`" meaning that they should be manipulated as a tuple of immediate values rather than being heap-allocated; this is suitable for small records or simple sum types. As a degenerate special case, a data type may have zero constructors; a `case` over such a value of such a type has no arms and thus arbitrary result type, and its dynamic semantics is to abort.

The semantics of RML declarations and expressions are straightforward, so we omit a formal presentation. As in SML, evaluation order is fixed left-to-right, and all conditional control flow is governed by `case` expressions. User-defined functions and primitives all receive their parameters by value. There is no built-in facility for exceptions, nor can these be sensibly implemented using call-by-value primitives.

The unit of translation is a *component*: a sequence of type and value declarations

[3] I.e., in a datatype definition abstracted over a given list of type variables, every right-hand-side mention of that datatype must be instantiated at exactly the same variables.

$$
\begin{array}{rll}
\textit{type} ::= & \texttt{type}\ K\ \texttt{"}\textit{string}\texttt{"} & \text{(primitive types)} \\
\textit{value} ::= & \texttt{val}\ k : K\ \texttt{"}\textit{string}\texttt{"} & \text{(primitive constants)} \\
| & \texttt{val}\ p(\langle v : \tau \rangle)\texttt{:}(v : \tau)\ [\texttt{pure}]\ \texttt{"}\textit{string}\texttt{"} & \text{(primitive functions)} \\
 & & \\
t ::= & \texttt{template}\ \textit{name}\ \langle \textit{type} \rangle\ \langle \textit{value} \rangle & \text{(templates)}
\end{array}
$$

Fig. 7. Template specification syntax. Types τ are as in RML.

(e.g., as in Figure 4). Each RML component has an `export` clause, which lists the types and values that are to be exported for use by 3GL components of the system and specifies 3GL names for them. In particular, the main program or driver for an executable is always written in the host 3GL, and invokes RML code via one or more of the exported functions. Polymorphic types and values can only be exported at specific monomorphic instances. Argument and result types of exported functions must be first-order. Formally, the "meaning" of a component is an environment mapping 3GL names to RML types and values; this environment must not be altered by the translation process.

Our translator currently does not directly support multiple RML components in a program, although functions generated from one RML component can be treated like any other 3GL functions and imported as (first-order) primitives into another RML component via the template mechanism. There are two obvious reasons why it might be useful to divide the RML code for a large system into multiple components: to provide independent namespaces (e.g., for libraries), or to speed up system building via separate compilation. We plan to extend our system to support the former goal, which should be straightforward. Separate compilation would be much harder, however, since many of our translation strategies depend fundamentally on having access to all the RML source code at one time.

4 Templates

Each RML component is translated with respect to a particular *template*, which specifies the interface between 3GL components and RML code. The template definition plays two key roles. It specifies which types and operators, implemented in the 3GL, are to be available to RML code as primitives; this information is used by the translator when parsing and type-checking RML components. The template also includes macro definitions for the types and operators in terms of 3GL code fragments; these are used by the translator when it generates 3GL code from RML. Templates are defined using a small special-purpose language, whose concrete syntax is shown in Figure 7. Template specifications make heavy use of quoted *strings*, which represent text in the target 3GL; they utilize a standard set of escape conventions based on those of SML. Figure 3 provides a typical example of an Ada template; a C template definition would have the same format, though of course the macro text would differ.

Primitive types typically include both general-purpose types (e.g., `integer`, `real`, ...) and application-specific types (e.g., `transform` or `point`). A primitive type is introduced by a `type` declaration, which gives the type a name to be used within RML code and specifies the corresponding 3GL type name—built-in or user-defined—that provides a concrete realization of the type. Primitive types are always monomorphic, i.e., not parameterized.

Primitive values and operators are defined by `val` declarations, which specify their types and their expansions into 3GL code. Values, operator arguments, and operator results must have primitive types,[4] which implies that values and operators are always monomorphic and first-order. A value declaration specifies the (RML) type of the value and the corresponding 3GL syntax for it.[5] An operator declaration specifies formal names and types for the operator's arguments and result; the corresponding 3GL code string is treated as a macro using the formal names as parameters. Formal parameters are referenced inside the string by surrounding them with back-quotes (‘). For example, the definition of the primitive division operator might be

```
val div (x0:real,x1:real) : (res:real) "‘res‘ := ‘x0‘ / ‘x1‘;"
```

An RML expression like `val a = div (x,2)` eventually leads to the Ada code

```
... a := x / 2; ...
```

As this example illustrates, the expansions for operators are statements rather than expressions, which permits more elaborate definitions. To make this possible from the RML side, code generation is performed on an imperative intermediate form (see Section 12.1) in which primitive operator calls appear only as the right-hand sides of assignment statements, so the result of an operation is "returned" by assigning it to a variable. All actual arguments to operators are either variable names or constants, which prevents potential problems with multiple uses of a formal argument in the macro.

Operators on general-purpose primitive types (e.g., the `div` operator described above) can often be implemented using built-in operators of the 3GL. Application-specific types and operators usually depend on non-trivial 3GL type definitions and library code. If desired, calls to small functions can be inlined by hand in the operator definition (e.g., `apply` in our example).[6] Operators marked as `pure` are assumed (without separate verification) to have no side-effects; the translator can apply more aggressive optimizations to expressions that involve only pure operators (see Section 8).

[4] There is also a mechanism, which we do not describe in detail here, for using the algebraic type `bool = true | false`; this permits RML code to perform conditional computation based on the result of a primitive operation.

[5] In principle, every integer, real, and string literal used in a RML program should be specified this way; to avoid this tedium, the template mechanism has all such literal constants "built-in."

[6] Our experience has been that 3GL compilers cannot be depended upon to perform such inlining automatically.

5 Compiler Architecture and Representations

The compiler is structured as a pipeline operating on a series of specialized, typed intermediate representations; see Figure 8. This section of the paper summarizes the most important steps in the compilation sequence, and serves as a guide to the detailed descriptions of these steps in the sections that follow.

- RML code is parsed from a concrete text representation or loaded from a binary representation produced by a separate generator tool. Parsing is performed with respect to a particular template definition, which provides a particular set of primitive types and operators.
- The RML code is annotated with type information using conventional Hindley-Milner type inference. The annotated code is then translated to monomorphic form (Section 6).
- The monomorphic RML code is transformed to a more restrictive language, called SIL (for "Sequentialized Intermediate Language"), which is a variant of A-normal form (Flanagan et al., 1993), closely related to continuation-passing style (Steele, 1978; Kranz et al., 1986; Appel, 1992). In SIL (Figure 12), all arguments to functions and primitives are required to be named variables or constants. Thus, the translation from RML to SIL (Section 7) effectively fixes the order of evaluation of all primitives. SIL also supports "jump points," i.e., locally scoped continuation functions (Kelsey, 1995), though the initial translation to SIL doesn't use these.
- The SIL code is optimized (Section 8) by repeated application of rewrite rules that encode "partial-evaluation style" improvements: value and variable propagation, simplification of `case` expressions over known values, elimination of dead code and unused datatypes, and conservative function inlining.
- The SIL code is reduced to first-order form (Section 9). The resulting code is then re-optimized (Section 10).
- All tail calls are changed into jumps, merging mutually recursive functions if necessary (Section 11).
- The SIL code is transformed into imperative target code, in two stages, which are treated only briefly in this paper (Section 12). First, SIL code is transformed into a further intermediate form, called MIL (for "Mutable Intermediate Language"), which abstracts the essential characteristics shared by C, Ada83, and similar languages. Then, MIL code is translated into Ada83 or C code using the template macros.

The entire compiler amounts to about 20,000 lines of Standard ML, and runs under version 109.31 of the Standard ML of New Jersey system.

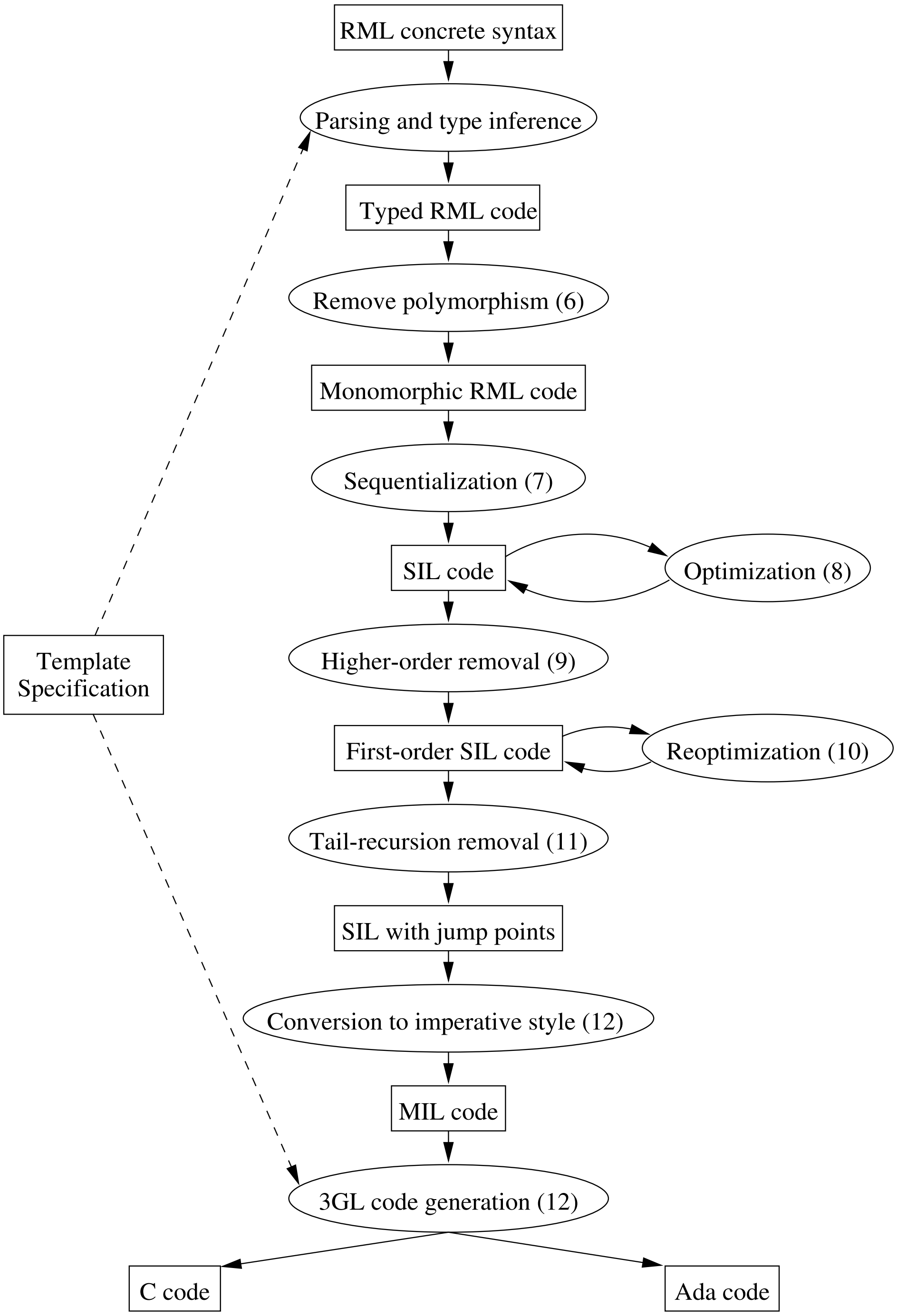

Fig. 8. Architecture of the compiler. Numbers in parentheses refer to section numbers in this paper where the relevant transformation is described.

6 Eliminating Polymorphism

6.1 Concept

Our target 3GLs do not directly support parametric polymorphism.[7] The translator therefore converts polymorphic components to monomorphic ones by producing specialized clones of polymorphic functions and constructors for each type at which they are used. By arranging to perform this step early in the compilation process, as an RML-to-RML translation, we clear the way for later transformation algorithms, notably the higher-order function remover and the representation analyzer, which require monomorphic input.

The specialization algorithm operates on the complete type-checked source program, in which every use of a polymorphic identifier has been annotated with its instantiated type. Given this representation, the full set of instantiations for each type abstraction can be enumerated, in a way which will be described below. RML's restrictions against polymorphic recursion in datatypes or functions guarantee that these sets are finite. Moreover, the complete set of instantiations for the bound type variables in a recursive function or datatype definition (or mutually recursive set of definitions) can always be determined without looking at the right-hand side(s) of the definition(s). This fact allows the instantiations to be enumerated by a one-pass algorithm that doesn't require a fixed-point calculation.

In our Section 2 example, the specializer generates two versions of the `list` datatype, specialized to `points` and `transforms` respectively, and two corresponding versions of the `foldl` function. The resulting component is shown in Figure 9.

6.2 Details of the Algorithm

The complete specialization algorithm consists of three passes over the type-annotated program produced by a standard inferencer. The first pass replaces any occurrences of *free* type variables by an arbitrary trivial type; this is safe because the computation never examines values whose types involve free type variables (Morrisett et al., 1995). The second pass computes a mapping from each polymorphic variable and algebraic type constructor to its corresponding set of instantiations. The third pass uses this mapping to perform the actual specialization.

The enumeration pass is by far the most complex of the three; details are given in Figure 10. To explain the algorithm, we first require some terminology. A (simultaneous) *substitution* $S = (\langle t\rangle \mapsto \langle\tau\rangle)$ is a mapping from a sequence of n type variables to a corresponding sequence of n types. Applying a substitution S to a type τ has the usual effect of replacing each type variable $t \in Dom(S)$ with $S(t)$, while leaving other type variables and all type constructors unchanged. We further define the result of applying a substitution S to a sequence of types $\langle\tau\rangle$ to be the sequence $\langle S(\tau)\rangle$. A *multi-substitution* M is a mapping $(\langle t\rangle \mapsto \{\langle\tau\rangle\})$ from a sequence

[7] Actually, Ada generics have the necessary power, but certain restrictions on the form of generic package interfaces can cause unnecessary extra copies of code to be generated.

```
export
    type plist "PList"
    val PNil : plist "PNil"
    val PCons : point * plist -> plist "PCons"
    val doit : plist -> plist "doit"

datatype  plist = PNil | PCons of point * PList
datatype  tlist = TNil | TCons of transform * TList

val rec foldl0 : (transform * transform -> transform) *
                                      transform * tlist -> transform =
  fn (c,n,l) =>
       case l of
         TNil => n
       | TCons(x,r) => foldl0 (c,c(x,n),r)

val rec foldl1 : (point * plist -> plist) * plist * plist -> plist =
  fn (c,n,l) =>
       case l of
         PNil => n
       | PCons(x,r) => foldl0 (c,c(x,n),r)

val ts : tlist =  TCons(...)

val rec doit : plist -> plist  =
  fn ps =>
      let val whole_t : transform =
                 foldl0(fn (t1,t2) => compose(t1,t2),id,ts)
      in let val consapp : point * plist -> plist =
                 fn (x,l) => PCons(apply(whole_t,x),l)
         in let val ps0 : plist = foldl1(consapp,PNil,ps)
            in foldl1(fn (x,l) => Cons(x,l),PNil,ps0)
```

Fig. 9. RML abstract syntax for geometric example component after type specialization. Most type annotations are omitted to improve readability.

of n type variables to a *set* of corresponding sequences of n types; it thus compactly describes a set of substitutions with a common domain. We define the result of applying M to a type (resp. a sequence of types) to be the *set* of types (resp. of sequences of types) resulting from applying the individual substitutions in turn and removing duplicates. Any substitution can be viewed as a multi-substitution by making the result into a singleton set. If $M_1 = (\langle t \rangle \mapsto T_1)$ and $M_2 = (\langle t \rangle \mapsto T_2)$ are multi-substitutions with the same domain, we define their *sum*, $M_1 \uplus M_2$, as the multi-substitution $(\langle t \rangle \mapsto T_1 \cup T_2)$, where $\cup$ represents ordinary set union with removal of duplicates. If $\langle M \rangle$ is a sequence of multi-substitutions all having the same domain, we write $\biguplus \langle M \rangle$ for their combined sum. We define the *composition* $M_2 \circ M_1$ of multi-substitutions $M_1 = (\langle t \rangle \mapsto T_1)$ and M_2 to be the multi-substitution $(\langle t \rangle \mapsto \bigcup \{M_2(\langle \tau \rangle) \mid \langle \tau \rangle \in T_1\})$, where $\bigcup$ computes the union of the members of a set of sets.

$$
\begin{aligned}
\mathcal{T}\llbracket K \rrbracket &= \emptyset \\
\mathcal{T}\left\llbracket (\langle \tau \rangle_{*}) \to \tau_0 \right\rrbracket &= (\biguplus \langle \mathcal{T}\llbracket \tau \rrbracket \rangle) \uplus \mathcal{T}\llbracket \tau_0 \rrbracket \\
\mathcal{T}\left\llbracket (\langle \tau \rangle_{,})\, D \right\rrbracket &= (\biguplus \langle \mathcal{T}\llbracket \tau \rrbracket \rangle) \uplus \{ D \mapsto (\mathit{Tyvarsof}\llbracket D \rrbracket \mapsto \langle \tau \rangle) \}
\end{aligned}
$$

$$
\mathcal{S}\left\llbracket \forall \langle t \rangle_{,} . \tau \right\rrbracket = \mathcal{T}\llbracket \tau \rrbracket
$$

$$
\begin{aligned}
\mathcal{E}\llbracket (k : K) \rrbracket &= \emptyset \\
\mathcal{E}\llbracket (v : \tau) \rrbracket &= \{ v \mapsto \mathit{Inst}(\mathit{Schemeof}\llbracket v \rrbracket, \tau) \} \uplus \mathcal{T}\llbracket \tau \rrbracket \\
\mathcal{E}\left\llbracket e_0\, (\langle e \rangle_{,}) \right\rrbracket &= \mathcal{E}\llbracket e_0 \rrbracket \uplus (\biguplus \langle \mathcal{E}\llbracket e \rrbracket \rangle) \\
\mathcal{E}\left\llbracket (c : \tau)\, (\langle e \rangle_{,}) \right\rrbracket &= \{ \mathit{Tyconof}\llbracket c \rrbracket \mapsto \mathit{Inst}(\mathit{Schemeof}\llbracket c \rrbracket, \tau) \} \\
&\quad \uplus \mathcal{T}\llbracket \tau \rrbracket \uplus (\biguplus \langle \mathcal{E}\llbracket e \rrbracket \rangle) \\
\mathcal{E}\left\llbracket p(\langle e \rangle_{,}) \right\rrbracket &= \biguplus \langle \mathcal{E}\llbracket e \rrbracket \rangle \\
\mathcal{E}\llbracket \mathtt{fn}\ \mathit{inl}\ \mathit{rule} \rrbracket &= \mathcal{R}\llbracket \mathit{rule} \rrbracket \\
\mathcal{E}\llbracket \mathtt{let}\ \mathit{vdecs}\ \mathtt{in}\ e \rrbracket &= \mathcal{D}(\mathcal{E}\llbracket e \rrbracket)\llbracket \mathit{vdecs} \rrbracket
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{E}\left\llbracket \mathtt{case}\ e\ \mathtt{of}\ \langle (c : \tau)\ \mathit{rule} \rangle_{|} \right\rrbracket = \\
&\qquad \mathcal{E}\llbracket e \rrbracket \uplus \left(\biguplus \langle \{ \mathit{Tyconof}\llbracket c \rrbracket \mapsto \mathit{Inst}(\mathit{Schemeof}\llbracket c \rrbracket, \tau) \} \uplus \mathcal{T}\llbracket \tau \rrbracket \uplus \mathcal{R}\llbracket \mathit{rule} \rrbracket \rangle \right)
\end{aligned}
$$

$$
\mathcal{R}\left\llbracket (\langle v : \tau \rangle_{,})\ \mathtt{=>}\ e \right\rrbracket = \langle \mathcal{T}\llbracket \tau \rrbracket \rangle \uplus \mathcal{E}\llbracket e \rrbracket
$$

$$
\begin{aligned}
\mathcal{D}(I)\llbracket \mathtt{val}\ v : \sigma\ \mathtt{=}\ e \rrbracket &= I \uplus (I\llbracket v \rrbracket \circ \mathcal{S}\llbracket \sigma \rrbracket) \\
&\quad \uplus \left(I\llbracket v \rrbracket \circ \left(\mathcal{E}\llbracket e \rrbracket \Big|_{\mathit{Free}\llbracket e \rrbracket} \right) \right) \uplus \left(\mathcal{E}\llbracket e \rrbracket \Big|_{\mathit{Bound}\llbracket e \rrbracket} \right)
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{D}(I)\left\llbracket \mathtt{val\ rec}\ \langle v : \sigma\ \mathtt{=}\ \mathtt{fn}\ \mathit{inl}\ \mathit{rule} \rangle_{\mathtt{and}} \right\rrbracket = \\
&\qquad I \uplus (M \circ (\biguplus \langle \mathcal{S}\llbracket \sigma \rrbracket \rangle)) \\
&\qquad \uplus \left(M \circ \left(\biguplus \left\langle \mathcal{R}\llbracket \mathit{rule} \rrbracket \Big|_{\mathit{Free}\llbracket \mathit{rule} \rrbracket} \right\rangle \right) \right) \\
&\qquad \uplus \left(\biguplus \left\langle \mathcal{R}\llbracket \mathit{rule} \rrbracket \Big|_{\mathit{Bound}\llbracket \mathit{rule} \rrbracket} \right\rangle \right) \\
&\qquad \text{where } M = \biguplus \langle I\llbracket v \rrbracket \rangle
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{DS}(I)\llbracket \langle \mathit{vdecs} \rangle\ \mathit{vdecs}_0 \rrbracket &= \mathcal{DS}(\mathcal{D}(I)\llbracket \mathit{vdecs}_0 \rrbracket)\llbracket \langle \mathit{vdecs} \rangle \rrbracket \\
\mathcal{DS}(I)\llbracket\ \rrbracket &= I
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{AS}(I)\llbracket \langle \mathit{atdecs} \rangle\ \mathit{atdecs}_0 \rrbracket &= \mathcal{AS}(\mathcal{A}(I)\llbracket \mathit{atdecs}_0 \rrbracket)\llbracket \langle \mathit{atdecs} \rangle \rrbracket \\
\mathcal{AS}(I)\llbracket\ \rrbracket &= I
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{A}(I)\left\llbracket \mathtt{datatype}\ \left\langle (\langle t \rangle_{,})\ D\ \mathit{flt}\ \mathtt{=}\ \left\langle \mathtt{c}\ [\mathtt{of}\ \langle \tau \rangle_{*}] \right\rangle_{|} \right\rangle_{\mathtt{and}} \right\rrbracket = \\
&\qquad I \uplus (I\llbracket D \rrbracket \circ (\biguplus \langle \biguplus \langle \biguplus \langle \mathcal{T}\llbracket \tau \rrbracket \rangle \rangle \rangle))
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{X}\llbracket \mathtt{type}\ \tau\ \mathtt{"}\mathit{name}\mathtt{"} \rrbracket &= \mathcal{T}\llbracket \tau \rrbracket \\
\mathcal{X}\llbracket \mathtt{val}\ v : \tau\ \mathtt{"}\mathit{name}\mathtt{"} \rrbracket &= \{ v \mapsto \mathit{Inst}(\mathit{Schemeof}\llbracket v \rrbracket, \tau) \} \uplus \mathcal{T}\llbracket \tau \rrbracket
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{C}\llbracket \mathtt{export}\ \langle \mathit{export} \rangle\ \langle \mathit{atdecs} \rangle\ \langle \mathit{vdecs} \rangle \rrbracket = \\
&\qquad \mathcal{AS}(\mathcal{DS}(\biguplus \langle \mathcal{X}\llbracket \mathit{export} \rrbracket \rangle)\llbracket \langle \mathit{vdecs} \rangle \rrbracket)\llbracket \langle \mathit{atdecs} \rangle \rrbracket
\end{aligned}
$$

Fig. 10. Enumeration of instances of polymorphic identifiers.

An *instantiation map* $I\,[\![x]\!]$ is a mapping from polymorphic identifiers $x : \forall\langle t\rangle_{,}.\tau_0$ to multi-substitutions with domain $\langle t\rangle$; we will build instantiation maps whose domains include both variables and algebraic type constructors. If I is an instantiation map and S is a set of identifiers, we write $I\,|_S$ for the instantiation map that results from restricting I's domain to S. If I_1 and I_2 are instantiation maps, we write $I_1 \uplus I_2$ for the instantiation map $\{x \mapsto I_1\,[\![x]\!] \uplus I_2\,[\![x]\!] \mid x \in (Dom(I_1) \cup Dom(I_2))\}$. Further, if $\langle I\rangle$ is a sequence of instantiation maps, $\biguplus \langle I\rangle$ represents their sequential combination under $\uplus$. Finally, if $I = \{x \mapsto M_x\}$ is an instantiation map and M is a multi-substitution, we define the composition $M \circ I$ to be the instantiation map $\{x \mapsto M \circ M_x\}$.

Each of the syntax-directed rules in Figure 10 maps a syntactic fragment to the instantiation map describing the sets of type instantiations induced by mentions of variables and constructors within that fragment. In particular, $\mathcal{C}$ calculates the instantiation map for an entire component, whose domain is the component's complete set of top-level and `let`-bound variables and algebraic type constructors. The algorithm walks over the component in bottom-up fashion, so that information about the (non-recursive) mentions of an identifier has always been incorporated into an instantiation map before the definition of that identifier is processed; this map is passed as an auxiliary argument I to the rule that processes the definition, i.e., $\mathcal{D}$ or $\mathcal{A}$. Thus, for example, to process a fragment `let val` $v : \forall\langle t\rangle_{,}.\tau$ `=` e_1 `in` e_2, the algorithm

i. builds an instantiation map I_{e_2} based on e_2;
ii. builds instantiation maps I_τ based on τ and I_{e_1} based on e_1, in which the instantiating types may mention the type variables $\langle t\rangle$;
iii. divides I_{e_1} into two parts, I_{Fe_1} and I_{Be_1}, corresponding to the free and bound variables of e_1;
iv. expands I_{Fe_1} to I'_{Fe_1} by pre-composing with $I_{e_2}\,[\![v]\!]$, the multi-substitution describing all possible instantiations for the $\langle t\rangle$;
v. similarly expands I_τ to I'_τ;
vi. sums I'_τ , I'_{Fe_1}, the (unexpanded) I_{Be_1}, and I_{e_2} to yield the final map for the overall `let` expression.

The distinction between free and bound variables effectively prevents the map entry for a locally-defined function from being refined further after its definition has been processed, which is important for the specialization pass. The rules for recursive function and datatype definitions are similar. Because RML prohibits polymorphic recursive definitions of functions or algebraic types, the auxiliary I is guaranteed to describe *all* instantiations of the identifier being defined; that is, there is no need to look at the right-hand side of the definition as well. However, in the recursive function case it *is* necessary to combine instance information about uses of *all* mutually-recursive functions before pre-composing.

The algorithm relies on a number of auxiliary functions. $Free[\![e]\!]$ returns the set of free variables and type constructors mentioned in expression or rule e; similarly, $Bound[\![e]\!]$ returns the set of bound variables of e. $Inst(\forall\langle t\rangle_{,}.\tau_0, \tau)$ returns a substitution $S = (\langle t\rangle \mapsto \langle \tau'\rangle)$ such that $S(\tau_0) = \tau$; it will only be called on argu-

```
datatype 'a p = P of 'a * 'a
val f:∀'b.'b -> 'b p = fn (x:'b) => (P:'b * 'b -> 'b p)(x:'b,x:'b)
val g:∀'c,'d.'c * 'd -> 'd p =
      fn (y:'c,z:'d) =>
         let val h:∀'e.'e -> 'd p = fn (w:'e) => (f:'d -> 'd p)(z:'d)
         in let val v:'c p = (f:'c -> 'c p)(y:'c)
            in (h:'c p -> 'd p)(v:'c p)
val a:bool p = (g:int * bool -> bool p) (3:int,true:bool)
val b:string p = (g:bool * string -> string p) (false:bool,"abc":string)
```

Fig. 11. Example of nested polymorphic functions.

ments for which the result substitution is guaranteed to exist. We also assume the existence of reconstruction functions *Schemeof*$[\![x]\!]$, which returns the (possibly degenerate) type-scheme corresponding to any variable or constructor x; *Tyconof*$[\![c]\!]$, which returns the algebraic type constructor to which data constructor c belongs; and *Tyvarsof*$[\![D]\!]$, which returns the (possibly empty) sequence of type variables over which the algebraic type constructor D is abstracted. Moreover, we assume certain consistency conditions on these functions: the schemes of any two mutually-recursive functions must have the same sequence of bound type variables; similarly, the schemes of any two data constructors of the same type constructor or of mutually-recursive type constructors must have the same sequence of bound type variables, which must also match the sequence(s) returned by *Tyvarsof* on the type constructor(s). These conditions are naturally met by the annotations produced by a standard type-inferencer, provided that all recursive definitions are separated into their strongly-connected components before inferencing.

As an (artificial) example of the algorithm's operation, consider the code in Figure 11, written in explicitly typed form. The computation proceeds roughly as follows (ignoring the generation of empty map entries for non-polymorphic variables):

- The declaration of `b` is processed, yielding an instantiation map

$$I_1 = \{\mathtt{g} \mapsto (\langle\mathtt{'c},\mathtt{'d}\rangle \mapsto \{\langle\mathtt{bool},\mathtt{string}\rangle\}), \mathtt{p} \mapsto (\langle\mathtt{'a}\rangle \mapsto \{\langle\mathtt{string}\rangle\})\}$$

- The declaration of `a` is processed, yielding an instantiation map

$$I_2 = \{\mathtt{g} \mapsto (\langle\mathtt{'c},\mathtt{'d}\rangle \mapsto \{\langle\mathtt{int},\mathtt{bool}\rangle\}), \mathtt{p} \mapsto (\langle\mathtt{'a}\rangle \mapsto \{\langle\mathtt{bool}\rangle\})\}$$

- I_2 is added to I_1 to produce the overall map for the declarations of `a` and `b`

$$I_3 = \left\{ \begin{array}{l} \mathtt{g} \mapsto (\langle\mathtt{'c},\mathtt{'d}\rangle \mapsto \{\langle\mathtt{bool},\mathtt{string}\rangle, \langle\mathtt{int},\mathtt{bool}\rangle\}), \\ \mathtt{p} \mapsto (\langle\mathtt{'a}\rangle \mapsto \{\langle\mathtt{bool}\rangle, \langle\mathtt{string}\rangle\}) \end{array} \right\}$$

- The `let val v` expression is processed, yielding (in several steps) the map

$$I_4 = \left\{ \begin{array}{l} \mathtt{f} \mapsto (\langle\mathtt{'b}\rangle \mapsto \{\langle\mathtt{'c}\rangle\}), \\ \mathtt{h} \mapsto (\langle\mathtt{'e}\rangle \mapsto \{\langle\mathtt{'c}\rangle\}), \\ \mathtt{p} \mapsto (\langle\mathtt{'a}\rangle \mapsto \{\langle\mathtt{'c}\rangle, \langle\mathtt{'d}\rangle\}) \end{array} \right\}$$

- The body of the definition of h is processed, yielding the map

$$I_5 = \{\mathtt{f} \mapsto (\langle\mathtt{'b}\rangle \mapsto \{\langle\mathtt{'d}\rangle\}), \mathtt{p} \mapsto (\langle\mathtt{'a}\rangle \mapsto \{\langle\mathtt{'d}\rangle\})\}$$

- The composition $I_4\,[\![\mathrm{h}]\!] \circ I_5$ is computed, yielding I_5 unchanged.
- I_5 is added to I_4 to produce the overall map for the body of the definition of g:

$$I_6 = \left\{ \begin{array}{l} \mathtt{f} \mapsto (\langle\mathtt{'b}\rangle \mapsto \{\langle\mathtt{'c}\rangle, \langle\mathtt{'d}\rangle\}), \\ \mathtt{h} \mapsto (\langle\mathtt{'e}\rangle \mapsto \{\langle\mathtt{'c}\rangle\}), \\ \mathtt{p} \mapsto (\langle\mathtt{'a}\rangle \mapsto \{\langle\mathtt{'c}\rangle, \langle\mathtt{'d}\rangle\}) \end{array} \right\}$$

- The composition $I_3\,[\![\mathrm{g}]\!] \circ (I_6 \big|_{\{\mathtt{f},\mathtt{p}\}})$ is computed, yielding the map:

$$I_7 = \left\{ \begin{array}{l} \mathtt{f} \mapsto (\langle\mathtt{'b}\rangle \mapsto \{\langle\mathtt{int}\rangle, \langle\mathtt{string}\rangle, \langle\mathtt{bool}\rangle\}), \\ \mathtt{p} \mapsto (\langle\mathtt{'a}\rangle \mapsto \{\langle\mathtt{int}\rangle, \langle\mathtt{string}\rangle, \langle\mathtt{bool}\rangle\}) \end{array} \right\}$$

- I_7 is added to $I_6 \big|_{\{\mathtt{h}\}}$ and then to I_3 to produce the overall map for the declarations of g, h, a, and b:

$$I_8 = \left\{ \begin{array}{l} \mathtt{g} \mapsto (\langle\mathtt{'c}, \mathtt{'d}\rangle \mapsto \{\langle\mathtt{bool}, \mathtt{string}\rangle, \langle\mathtt{int}, \mathtt{bool}\rangle\}), \\ \mathtt{h} \mapsto (\langle\mathtt{'e}\rangle \mapsto \{\langle\mathtt{'c}\rangle\}), \\ \mathtt{f} \mapsto (\langle\mathtt{'b}\rangle \mapsto \{\langle\mathtt{int}\rangle, \langle\mathtt{string}\rangle, \langle\mathtt{bool}\rangle\}), \\ \mathtt{p} \mapsto (\langle\mathtt{'a}\rangle \mapsto \{\langle\mathtt{int}\rangle, \langle\mathtt{string}\rangle, \langle\mathtt{bool}\rangle\}) \end{array} \right\}$$

- Processing the definitions of f and p will produce the same map I_8.

I_8 can now be used to guide the the specialization pass of the algorithm. Two specialized copies are made of function g, corresponding to the two instantiations for $\langle$'c, 'd$\rangle$. Note the importance of tracking the *sequences* of instantiations for these type variables; if the instantiations of each variable were tracked separately, there would be no way to distinguish the correct instantiations from the spurious ones with 'c = bool,'d = bool and 'c = int, 'd = string. Within each copy of g, a single specialized version is made of h, with 'e instantiated to the relevant instance of 'c, namely int within the first copy of g and bool within the second. Note that if the enumeration algorithm did not "freeze" the instantiation map for h at its point of definition, the final map would have the entry

$$\mathtt{h} \mapsto (\langle\mathtt{'e}\rangle \mapsto \{\langle\mathtt{int}\rangle, \langle\mathtt{bool}\rangle\})$$

While this would correctly enumerate the versions of h that are required, it would fail to indicate that only *one* instance is needed within each copy of g, nor say which one is needed where. Finally, three specialized copies are made of p and f, corresponding to the three instantiations for 'a and for 'b. We omit a detailed description of the specialization pass, which is quite straightforward given the existence of the instantiation map.

6.3 Discussion

In the worst case, the size of the monomorphic program produced by this algorithm may be exponential in the size of the original polymorphic program. However, we have not found code explosion to be a serious problem in practice, as most polymorphic functions tend to be small; this is probably because the more polymorphic a function is, the fewer useful things it can do (Wadler, 1989)!

The idea of removing parametric polymorphism by specialization has received much informal discussion, and a small experiment has been attempted for Gofer (Jones, 1993), but we are not aware of any previous practical compiler based on this approach. Analysis of benchmarks run on the Til compiler (Tarditi et al., 1996) indicates that the compiler removes essentially all polymorphism as the result of aggressive function inlining, thus offering independent evidence that specialization need not lead to excessive code explosion. However, since Til does not *guarantee* to produce a monomorphic program, it cannot take full advantage of having one during later compilation stages, as our translator does.

7 Sequentialization

RML has a rich collection of expression forms; our target 3GLs have severely limited expressions. Also, even where there appears to be a direct correspondence between expression forms in RML and a target language, evaluation order may differ. Thus, the first step in translating RML is to simplify expressions and name all intermediate results, at the same time explicitly sequentializing the computation in the intended order. We call the resulting language SIL (for "Sequentialized Intermediate Language"); its syntax is specified in Figure 12. Compared with RML, the most important differences are that arguments to applications and discriminants in `case` expressions must be *simple,* i.e., variables or constants, and there are no anonymous function expressions. SIL's type system is monomorphic, since any polymorphism has already been removed at the RML level. This means that types can no longer mention type variables, there are no more type schemes, and type annotations are dropped wherever they have become redundant (e.g., on variable mentions); exports and algebraic type declarations are otherwise identical to RML. Jump points (`label` and `goto`) do not appear in the initial translations of RML code; their use is discussed in Section 11. As an example, Figure 13 shows the SIL form of the `doit` functions from the monomorphic version (Figure 9) of our running example from Section 2.

The transformation from RML to SIL essentially performs the naming and sequentialization steps of a continuation-passing-style (CPS) transform (Steele, 1978; Kranz et al., 1986; Appel, 1992), but without introducing full-scale continuations. We omit the details of the transformation, which are fairly straightforward. The resulting SIL code closely resembles other "almost-CPS" forms that have been adopted in recent functional language compilers (Lawall and Danvy, 1993; Flanagan et al., 1993; Kelsey, 1995; Tarditi, 1996).

SIL adopts a relatively permissive approach to the location of `let`-bindings: it permits the result of a `case` to be `let`-bound, unlike A-normal form (Flanagan et al., 1993), and even permits the result of a `let` expression to be `let`-bound, unlike both A-normal form and Til's B-form (Tarditi, 1996). This extra flexibility makes it easy to transform RML `case` expressions into SIL without duplicating code

$\tau ::=$	K	(primitive types)
$\mid$	D	(monomorphic algebraic types)
$\mid$	$(\langle\tau\rangle_*) \to \tau$	(function types)
$se ::=$	$(k : K)$	(primitive constants)
$\mid$	v	(variables)
$e ::=$	se	(simple expressions)
$\mid$	$v(\langle se\rangle_,)$	(function applications)
$\mid$	$c(\langle se\rangle_,)$	(constructor applications)
$\mid$	$p(\langle se\rangle_,)$	(primitive applications)
$\mid$	$\texttt{let}\ decs\ \texttt{in}\ e$	(local declarations)
$\mid$	$\texttt{case}\ se\ \texttt{of}\ \left\langle c(\langle v\rangle_,)\ \texttt{=>}\ e\right\rangle_{\mid}$	(destructuring)
$\mid$	$\texttt{goto}\ l(\langle se\rangle_,)$	(jumps to local labels)
$vdec ::=$	$\texttt{val}\ v\texttt{:}\tau\ \texttt{=}\ e$	(variable declarations)
$fdecs ::=$	$\texttt{fun}\ \left\langle v\ [\texttt{inline}]\ (\langle v : \tau\rangle_,)\texttt{:}\tau\ \texttt{=}\ \texttt{e}\right\rangle_{\texttt{and}}$	(mutually recursive function declarations)
$ldecs ::=$	$\texttt{label}\ \left\langle l(\langle v : \tau\rangle_,)\texttt{:}\tau\ \texttt{=}\ e\right\rangle_{\texttt{and}}$	(mutually recursive jump point declarations)
$decs ::=$	$vdec$	(variable declarations)
$\mid$	$fdecs$	(function declarations)
$\mid$	$ldecs$	(jump-point declarations)
$topdecs ::=$	$vdec$	(top-level variable declarations)
$\mid$	$fdecs$	(top-level function declarations)
$atdec ::=$	$D\ [\texttt{flat}]\ \texttt{=}\ \left\langle c\ [\texttt{of}\ \langle\tau\rangle_*]\right\rangle_{\mid}$	(algebraic type declarations)
$atdecs ::=$	$\texttt{datatype}\ \langle atdec\rangle_{\texttt{and}}$	(mutually recursive declarations)
$export ::=$	$\texttt{type}\ \tau\ \texttt{"}name\texttt{"}$	(type exports)
$\mid$	$\texttt{val}\ v : \tau\ \texttt{"}name\texttt{"}$	(value exports)
$comp ::=$	$\texttt{export}\ \langle export\rangle\ \langle atdecs\rangle\ \langle topdecs\rangle$	(components)

Fig. 12. SIL syntax.

```
fun doit (ps:plist) : plist =
  let val whole_t : transform =
            let fun comp (t1:transform,t2:transform) = compose(t1,t2)
            in foldl0(comp,id,ts)
  in let fun consapp (x:point,l:plist) =
            let val p0 = apply(whole_t,x)
            in PCons(p0,l)
     in let val ps0 : plist = foldl1(consapp,PNil,ps)
        in let fun cons (x:point,l:plist) = PCons(x,l)
           in foldl1(cons,PNil,ps0)
```

Fig. 13. SIL translation of geometric example component (selection).

or introducing continuation functions, and also keeps SIL closed under a larger class of optimization transformations.

8 Optimization

SIL code is optimized by repeated application of rewrite rules that encode "partial-evaluation style" improvements. These include propagation of simple expressions (constants and variables), simplification of `case` expressions over known values,[8] elimination of unused function and (pure) value bindings, elimination of unused datatypes, hoisting out of `let` bindings (described below), and conservative function inlining. A function application is inlined if

- it is the sole application of that function; or
- its body is "small, " i.e., a value, variable, or another application; or
- its body has the form of a case expression over an argument, the argument is a known value, and the relevant arm of the case is "small" (we call this *case splitting*); or
- the programmer demands inlining via a source pragma on the function definition.

To guarantee termination of the inliner, a function is never inlined into its own body. Our choice and implementation of optimizations was largely inspired by Appel and Jim (1998). The optimizer does not perform speculative inlining. Optimization passes are performed repeatedly until no change is observed or some fixed small number of passes has been reached. The optimization passes are preceded by a single round of eta-expansion to improve opportunities for inlining.

When a variable is `let`-bound to a `case` expression, it's value is not known at compile time, and so cannot be propagated. Hoisting `case` expressions out of `lets` is an optimization-enabling transform that can increase the amount of information available for constant propagation in each case arm. The general form of the transformation is:

[8] This optimization includes selection of fields from records with known values as an important special case.

```
let val v = case e0 of                          case e0 of
                C1(⟨w⟩,) => e1                    C1(⟨w⟩,) => let val v = e1 in e
              | C2(⟨w⟩,) => e2        ⟹         | C2(⟨w⟩,) => let val v = e2 in e
              | ...                             | ...
              | Cn(⟨w⟩,) => en                  | Cn(⟨w⟩,) => let val v = en in e
in e
```

It is primarily worth doing if e has the form $f(v)$ and it is possible to perform case splitting on f. In general, this is a dangerous transformation, since it duplicates the code for e in each case arm, so it performed only when e is a "small" expression. In addition, `let`s are always hoisted out of `let`s, as this never causes code explosion, and may help optimization by exposing more `case` hoisting opportunities.

The optimizations described above do not make essential use of type information, but our system does perform some simple type-based global optimizations. All uses of a "transparent" datatype of the form `datatype t = T of` τ can be replaced by direct uses of τ, and the datatype definition itself can then be removed. All uses of a "unit" datatype of the form `datatype t = T` in function or constructor argument lists can be eliminated (even for escaping functions), and any remaining values of type `t` can be replaced by the literal `T`. These forms of datatype quite often arise in code generated by the higher-order removal process (see Section 9).

Since RML has strict semantics, and templates may include impure operators, the optimizer must guarantee not to duplicate, reorder, or eliminate calls to primitives or to potentially nonterminating functions. In fact, none of the transformations described above induce duplication or reordering, and only "pure" expressions can be eliminated. Pure primitive operators are marked as such in the template definition; for simplicity, all user function calls are treated as impure. A more sophisticated approach would be to perform an effects analysis on functions to increase the number of eliminable expressions (e.g., (Talpin and Jouvelot, 1992; Tarditi, 1996)).

9 Removing Higher-order Functions

9.1 Concepts

Our target 3GLs do not directly support first-class nested functions; Ada83 does not even support pointers to top-level functions, and ANSI C does not support nested functions. We therefore must convert higher-order programs into equivalent first-order programs without nested functions, i.e., perform closure conversion. For simplicity, we wish to express the first-order programs in a strict subset of the original language, as in "closure-passing style" (Appel and Jim, 1989), where closures are represented as ordinary records, and are constructed and accessed using ordinary record operators. In particular, this would allow us to optimize closure manipulation operations using ordinary record optimizations. However, we would also like the closure-converted program to be well-typed according to the rules of the original language—rules that should also be enforceable in C or Ada. The difficulty is that two functions with the same type might well differ in the number and types of their free variables, and hence have closure records of completely different (structural) type.

Minamide, Morrisett, and Harper (1996) have treated this problem, but their solutions rely either on new language primitives for closure manipulation, which complicate subsequent optimization, or on giving closures existential types, a substantial complication to the compiler's type system. Neither solution leads to typable C or Ada. Moreover, both solutions continue to make use of (top-level) function pointers.

We take a different approach, which relies on having the whole monomorphic program available for analysis and transformation. It derives from the interpretive technique introduced by Reynolds (1972) and Warren (1982) and explored in typed settings by Bell, Bellegarde and Hook (Bell, 1994; Bellegarde and Hook, 1994; Bell et al., 1997). The key idea is to represent function closures as members of a *closure* algebraic data type (i.e., discriminated union). There is one constructor for each function definition in the program; its arguments are the function's free variables.[9] To convert a program to first order, all higher-order operations on functions are replaced by equivalent operations on closure values. Function definitions are lambda-lifted, and their original definitions are replaced by closure constructor applications; calls to "unknown" (i.e., lambda-bound) functions are transformed into calls to a "dispatch" function, passing a closure value as argument. The dispatch function examines the closure tag and passes control to the appropriate (lambda-lifted) function. 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 construction of a closure datatype value will be removed altogether by the standard dead-code elimination optimization. Figure 14 provides a simple example involving the higher-order function `twice`.

In a strongly-typed setting, a single closure datatype and dispatch function typically do not suffice: there must be a pair of them for each distinct arrow type in the program.[10] The translation algorithm must choose the correct dispatch function at each site by inspecting the type of the (original) function. This effect is illustrated by the code in Figure 15, which shows the closure-converted version of our running example.

In contrast with higher-order removal techniques based on code specialization (Chin and Darlington, 1996), our algorithm can handle programs in which the number of generated closures cannot be statically bounded. Figure 16 gives an example, based on the well-known encoding of an integer set as a characteristic function of type `int->bool`. Executing `upto`(n) builds the set $\{1,2,3,\ldots,n\}$, represented by $n+1$ `int->bool` closures, each (except the last) containing another such closure as a free variable. This pattern is reflected in the fact that type `clos` is recursive; in fact, it is isomorphic to the standard `list` datatype one might use to represent sets non-functionally!

[9] We use a *flat* closure representation in this paper; more elaborate representations could be handled in the same framework.

[10] It is not possible to build a single, polymorphic `dispatch` function using the ordinary typing rules for `case`.

Original code:

```
fun twice (f:int->int, i:int) : int = f(f i)
fun main (y:int,z:int) =
  let fun g1 (x:int) : int = +(x,y)
  in let fun g2 (x:int) : int = +(x,2)
     in ...twice(g1,z)...twice(g2,z)...
```

After closure-converting g1 *and* g2*:*

```
datatype clos = G1 of int | G2                          for type int->int
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 =                     for type int->int
     case c of
       G1 y => g1'(i,y)
     | G2 => g2' i

fun twice (f:clos, i:int) : int = dispatch(f,dispatch(f,i))
fun main (y:int,z:int) =
  let val g1 : clos = G1 y                              closure values
  in let val g2 : clos = G2
     in ...twice(g1,z)...twice(g2,z)...
```

Fig. 14. Simple example of typed closure conversion. The converted code is a slightly optimized version of the output produced by the formal algorithm in Section 9.2.

9.2 Details of the Algorithm

The core of the algorithm is a syntax-directed translation of terms to terms, under which

- each distinct arrow type is converted to a unique corresponding closure datatype;
- each function definition is "lambda-lifted" by augmenting its argument list with new arguments representing the function's free variables;
- these augmented functions are renamed and their definitions are lifted to top-level;
- each original function definition in the body of the program is replaced by a binding to an application of a freshly chosen closure constructor to the free variables;
- variables bound to function values become variables bound to closure values;
- calls to unknown functions become calls to the appropriate `dispatch` function, passing the closure datatype value as an extra argument;
- calls to known functions become calls to the corresponding lifted function, passing the free variables as extra arguments.

Along the way, the conversion keeps track of the new closure datatypes and data constructors, which are created incrementally; when all top-level declarations in the component have been converted, this information is used to construct the definitions

```
datatype tclos = Ccomp              for type transform*transform->transform
datatype pclos = Ccons | Cconsapp of transform
                                    for type point*plist->plist

fun comp' (t1:transform,t2:transform) : transform = compose(t1,t2)
and tdispatch (c:tclos,t1:transform,t2:transform) : transform =
      case c of                     for type transform*transform->transform
        Ccomp => comp'(t1,t2)
and consapp' (x:point,l:plist,whole_t:transform) : plist =
      let val p0 = apply(whole_t,x)
      in PCons(p0,l)
and cons' (x:point,l:plist) : plist = PCons(x,l)
and pdispatch (c:pclos,x:point,l:plist) : plist =
      case c of                     for type point*plist->plist
        Ccons => cons'(x,l)
      | Cconsapp (whole_t) => consapp' (x,l,whole_t)

fun foldl0 (c:tclos,n:transform,l:tlist) : transform =
      case l of
        TNil => n
      | TCons(x,r) =>
          let val n' : transform = tdispatch(c,x,n)
             in foldl0(c,n',r)

fun foldl1 (c:pclos,n:plist,l:plist) : plist =
       ...same except with val n' : plist = pdispatch(c,x,n)...

fun doit' (ps:plist,id:transform,ts:tlist) : plist =
      let val comp : tclos = Ccomp
          in let val whole_t : transform = foldl0(comp,id,ts)
              in let val consapp : pclos = Cconsapp(whole_t)
                 in let val ps0 : plist = foldl1(consapp,PNil,ps)
                    in let val cons : pclos = Ccons
                       in foldl1(cons,PNil,ps0)

val ts : tlist = TCons(...)

fun doit (ps:plist) = doit'(ps,id,ts)
```

Fig. 15. Geometric example component after closure-conversion. This is a slightly optimized version of the output produced by the formal algorithm in Section 9.2.

Original code:

```
fun empty (x:int) : bool = false
fun member (s:int->bool,x:int) : bool = s x
fun insert (s:int->bool,x:int) : int->bool =
     let fun s1 y = case =(x,y) of true => true | false => s x
     in s1
fun upto (n:int) : int->bool =
    case =(n,0) of true => empty | false => insert(upto(-(n,1),n))
fun main (n:int) : bool = lookup(upto n,100)
```

After closure-converting the `int->bool` *functions:*

```
datatype clos = E | S1 of int * clos               for type int->bool
fun empty' (x:int) : bool = false                  lambda-lifted functions
and s1' (y:int,x:int,s:clos) =
      case = (x,y) of true => true | false => dispatch(s,x)
and dispatch(c:clos,i:int) : bool =                for type int->bool
      case c of
        E => empty' i
      | S1 (x,s) => s1'(i,x,s)
val empty : clos = E                               closure value
fun member (s:clos,x:int) : bool = dispatch(s,x)
fun insert (s:clos,x:int) : clos =
      let val s1 : clos = S1(x,s)                  closure value
      in s1
fun upto (n:int) : clos = ... unchanged...
fun main (n:int) : bool = ... unchanged...
```

Fig. 16. Closure conversion of code for sets represented by characteristic functions. To improve readability, RML concrete syntax is used rather than SIL. The converted code is a slightly optimized version of the output produced by the formal algorithm in Section 9.2.

of the closure datatypes and the corresponding `dispatch` functions. Finally, these definitions are combined with the lifted function definitions and the converted terms to form the fully converted component definition.

As described here, the algorithm redefines every function-valued identifier as a closure and changes every arrow type to a closure type. But the signatures of exported values must not be changed; this implies that the argument and result types of exported functions must not involve arrow types (as noted in Section 3), and that exported functions themselves must not be closure converted. In practice, the system handles the latter problem by creating special *stub* versions of exported functions, with unchanged signatures, but we omit this detail from the formal presentation here, which is therefore correct only for programs that do not export functions.

A detailed specification of the term conversion algorithm is given in Figure 17. The translation of type τ is denoted $\overline{\tau}$. $\mathcal{TS}$ translates top-level declarations, $\mathcal{E}$

$$
\begin{array}{rcl}
\overline{K} & = & K \\
\overline{D} & = & D \\
\overline{(\langle\tau\rangle_{*}) \to \tau_0} & = & \mathit{ClosType}\left[\!\left[\,(\langle\overline{\tau}\rangle_{*}) \to \overline{\tau_0}\right]\!\right]
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{E}_k\,[\![se]\!] & = & se \\
\mathcal{E}_k\left[\!\left[v\,(\langle se\rangle_{,})\right]\!\right] & = & \mathit{if}\ v \in \mathit{Dom}(k) \\
 & & \mathit{then\ let}\ \langle fv\rangle\ =\ k\,[\![v]\!]\ \mathit{in}\ \mathit{Lift}[\![v]\!]\,(\langle se\rangle_{,},\langle fv\rangle_{,}) \\
 & & \mathit{else}\ (\mathit{Dispatch} \circ \mathit{ClosType} \circ \mathit{Typeof})\,[\![v]\!]\,(v\,,\langle se\rangle_{,}) \\
\mathcal{E}_k\left[\!\left[c\,(\langle se\rangle_{,})\right]\!\right] & = & c\,(\langle se\rangle_{,}) \\
\mathcal{E}_k\left[\!\left[p\,(\langle se\rangle_{,})\right]\!\right] & = & p\,(\langle se\rangle_{,}) \\
\mathcal{E}_k\,[\![\mathtt{let\ val}\ v : \tau\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2]\!] & = & \mathtt{let\ val}\ v : \overline{\tau}\ \mathtt{=}\ \mathcal{E}_k\,[\![e_1]\!]\ \mathtt{in}\ \mathcal{E}_k\,[\![e_2]\!] \\
\mathcal{E}_k\left[\!\left[\mathtt{let\ fun}\ \langle \mathit{fdec}\rangle_{\mathtt{and}}\ \mathtt{in\ e}\right]\!\right] & = & \mathit{let}\ \langle fv\rangle\ =\ \mathcal{FV}_k\left[\!\left[\mathtt{fun}\ \langle \mathit{fdec}\rangle_{\mathtt{and}}\right]\!\right]\ \mathit{in} \\
 & & \mathit{let}\ k'\ =\ k\ \langle +\ (\mathit{FunName}[\mathit{fdec}]\!]\mapsto \langle fv\rangle)\rangle\ \mathit{in} \\
 & & \mathcal{FS}_{k'}(\langle fv\rangle)\,[\![\langle \mathit{fdec}\rangle]\!]\,[\![e]\!]
\end{array}
$$

$$
\mathcal{E}_k\left[\!\left[\mathtt{case}\ se\ \mathtt{of}\ \left\langle c\,(\langle v\rangle_{,})\ \mathtt{=>}\ e\right\rangle_{|}\right]\!\right] = \mathtt{case}\ se\ \mathtt{of}\ \left\langle c\,(\langle v\rangle_{,})\ \mathtt{=>}\ \mathcal{E}_k\,[\![e]\!]\right\rangle_{|}
$$

$$
\begin{array}{rcl}
\mathcal{FS}_k(\langle fv\rangle)\,[\![\mathit{fdec}\,\langle \mathit{fdec}\rangle]\!]\,[\![e]\!] & = & \mathtt{let}\ \mathcal{F}_k(\langle fv\rangle)\,[\![\mathit{fdec}]\!]\ \mathtt{in}\ \mathcal{FS}_k(\langle fv\rangle)\,[\![\langle \mathit{fdec}\rangle]\!]\,[\![e]\!] \\
\mathcal{FS}_k(\langle fv\rangle)\,[\![\ ]\!]\,[\![e]\!] & = & \mathcal{E}_k\,[\![e]\!]
\end{array}
$$

$$
\begin{array}{l}
\mathcal{F}_k(\langle fv\rangle)\left[\!\left[v_0\ \ \mathit{inl}\ \ (\langle v : \tau\rangle_{,})\ \ \mathtt{:}\ \ \tau_0\ \ \mathtt{=}\ \ e\right]\!\right] = \\
\qquad \mathit{Lifted} := \mathit{Lifted} + \left(\mathit{Lift}[\![v_0]\!]\ \ \mathit{inl}\ \ (\langle v : \overline{\tau}\rangle_{,}\,,\left\langle fv : \overline{\mathit{Typeof}[\![fv]\!]}\right\rangle_{,})\,\mathtt{:}\,\overline{\tau_0}\ \mathtt{=}\ \mathcal{E}_k\,[\![e]\!]\right); \\
\qquad \mathit{let}\ tc\ =\ (\mathit{ClosType} \circ \mathit{Typeof})\,[\![v_0]\!]\ \mathit{in} \\
\qquad \mathit{let}\ c\ =\ \mathit{newDataCon}()\ \mathit{in} \\
\qquad \mathit{ClosData} := \mathit{ClosData} + \left(tc \mapsto \left(c, \mathit{Lift}[\![v_0]\!], \left\langle fv : \overline{\mathit{Typeof}[\![fv]\!]}\right\rangle\right)\right); \\
\qquad \mathtt{val}\ v_0 : tc\ \mathtt{=}\ c\,(\langle fv\rangle_{,})
\end{array}
$$

$$
\mathcal{A}\left[\!\left[\mathtt{datatype}\ \left\langle D\ \ \mathit{flt}\ \mathtt{=}\ \left\langle c\ \ [\mathtt{of}\ \ \langle\tau\rangle_{*}]\right\rangle_{|}\right\rangle_{\mathtt{and}}\right]\!\right] = \left\langle D\ \ \mathit{flt}\ \mathtt{=}\ \left\langle c\ \ [\mathtt{of}\ \ \langle\overline{\tau}\rangle_{*}]\right\rangle_{|}\right\rangle
$$

$$
\begin{array}{rcl}
\mathcal{TS}_k\,[\![\mathtt{val}\ v : \tau\ \mathtt{=}\ e\ \ \mathit{topdecs}]\!] & = & \mathtt{val}\ v : \overline{\tau}\ \mathtt{=}\ \mathcal{E}_k\,[\![e]\!]\ \ \mathcal{TS}_k\,[\![\mathit{topdecs}]\!] \\
\mathcal{TS}_k\left[\!\left[\mathtt{fun}\ \langle \mathit{fdec}\rangle_{\mathtt{and}}\ \ \mathit{topdecs}\right]\!\right] & = & \mathit{let}\ \langle fv\rangle\ =\ \mathcal{FV}_k\left[\!\left[\mathtt{fun}\ \langle \mathit{fdec}\rangle_{\mathtt{and}}\right]\!\right]\ \mathit{in} \\
 & & \mathit{let}\ k'\ =\ k\ \langle +\ (\mathit{FunName}[\mathit{fdec}]\!]\mapsto \langle fv\rangle)\rangle\ \mathit{in} \\
 & & \langle \mathcal{F}_{k'}(\langle fv\rangle)\,[\![\mathit{fdec}]\!]\rangle\ \ \mathcal{TS}_k\,[\![\mathit{topdecs}]\!] \\
\mathcal{TS}_k\,[\![\ ]\!] & = &
\end{array}
$$

Fig. 17. Closure conversion of SIL terms.

translates expressions, $\mathcal{F}$ translates functions, and $\mathcal{FS}$ is an auxiliary function for translating recursive sets of functions. Each of these translations is explicitly parameterized by an environment k that records those identifiers in the current scope that refer to known functions; where defined, $k\,[\![v]\!]$ returns the sequence of free variables of v, which are guaranteed to be in the current scope as well. $\mathcal{F}$ and $\mathcal{FS}$ are also parameterized by the function's sequence of free variables. $\mathcal{A}$ translates mutually recursive sets of algebraic type declarations.

In addition to producing result terms, these translations use side-effects to build important auxiliary structures:

$$
\begin{array}{rcl}
\mathcal{FV}_k\left[\!\left[\texttt{fun}\ \langle \mathit{fdec}\rangle_{\texttt{and}}\right]\!\right] & = & \bigcup\langle \mathcal{FV}_k\,[\![\mathit{fdec}]\!]\rangle \\
\mathcal{FV}_k\left[\!\left[v_0\ \ \mathit{inl}\ \ (\langle v:\tau\rangle_{,}\,)\ :\ \tau_0\ =\ e\right]\!\right] & = & \mathcal{FV}_k\,[\![e]\!] - \bigcup\langle\{v\}\rangle - \{v_0\} \\
\mathcal{FV}_k\,[\![(k:K)]\!] & = & \emptyset \\
\mathcal{FV}_k\,[\![v]\!] & = & \{v\} \\
\mathcal{FV}_k\left[\!\left[v\,(\langle se\rangle_{,}\,)\right]\!\right] & = & \bigcup\langle \mathcal{FV}_k\,[\![se]\!]\rangle \\
 & & \cup(\mathit{if}\ v\in \mathit{Dom}(k)\ \mathit{then}\ \{k\,[\![v]\!]\}\ \mathit{else}\ \{v\}) \\
\mathcal{FV}_k\left[\!\left[c\,(\langle se\rangle_{,}\,)\right]\!\right] & = & \bigcup\langle \mathcal{FV}_k\,[\![se]\!]\rangle \\
\mathcal{FV}_k\left[\!\left[p\,(\langle se\rangle_{,}\,)\right]\!\right] & = & \bigcup\langle \mathcal{FV}_k\,[\![se]\!]\rangle \\
\mathcal{FV}_k\,[\![\texttt{let val}\ v:\tau\ \texttt{=}\ e_1\ \texttt{in}\ e_2]\!] & = & \mathcal{FV}_k\,[\![e_1]\!]\cup(\mathcal{FV}_k\,[\![e_2]\!]-\{v\}) \\
\mathcal{FV}_k\left[\!\left[\texttt{let fun}\ \langle \mathit{fdec}\rangle_{\texttt{and}}\ \texttt{in}\ e\right]\!\right] & = & \mathcal{FV}_k\left[\!\left[\texttt{fun}\ \langle \mathit{fdec}\rangle_{\texttt{and}}\right]\!\right] \\
 & & \cup(\mathcal{FV}_k\,[\![e]\!]-\langle \mathit{FunName}[\![\mathit{fdec}]\!]\rangle) \\
\mathcal{FV}_k\,[\![\texttt{case}\ se\ \texttt{of}\ \langle c(\langle v\rangle)\ \texttt{=>}\ e\rangle]\!] & = & \mathcal{FV}_k\,[\![se]\!]\cup(\bigcup\left\langle \mathcal{FV}_k\,[\![e]\!]-\bigcup\langle\{v\}\rangle\right\rangle)
\end{array}
$$

Fig. 18. Calculation of free variables. The notation $\bigcup\langle X\rangle$ denotes the set union of all the sets X resulting from a calculation on members of a sequence.

i. a mapping *Lift* from source function names to corresponding lifted function names;
ii. a bijective mapping *ClosType* from source arrow types to corresponding closure datatype names;
iii. a mapping *Dispatch* from closure datatype names to corresponding dispatch function names;
iv. a set *Lifted* of lifted function definitions; and
v. a mapping *ClosData* from closure datatype names tc to sets of tuples

$$(dc, f, \langle fv:\tau\rangle)$$

where dc is a closure data constructor of tc, f is the corresponding (lifted) function name, and $\langle fv:\tau\rangle$ is the sequence of the corresponding function's free variables and their types.

The mappings *Lift*, *ClosType*, and *Dispatch* are treated as self-memoizing functions: they generate and return a fresh name when called with a given argument for the first time; subsequent calls with that argument return the same result as the first call. We also assume auxiliary functions *NewDataCon()*, which returns a fresh closure data constructor name each time it is called; *Typeof*$[\![e]\!]$, which reconstructs the (original) type of any source term e; and *FunName*$[\![f]\!]$, which extracts the function name from a declaration f. The *Lifted* set and *ClosData* sets are extended explicitly as a side-effect of the $\mathcal{F}$ translation. When the term translation is complete, these sets are used to generate the closure datatype definitions and dispatch functions, as described below. Note that the order in which side-effects are executed to build these structures does not alter the results except for choice of names, so the translation functions in Figure 17 do not have to be read with any particular imperative evaluation order in mind. For simplicity, the figure omits certain variable renamings required to maintain identifier uniqueness.

The auxiliary function $\mathcal{FV}_k\,[\![e]\!]$, specified in Figure 18, computes the free variables of expression e assuming the initial known function environment k. As specified,

$$
\begin{array}{l}
\mathcal{C}\,[\![\texttt{export}\ \langle export\rangle\ \langle atdecs\rangle\ \langle topdecs\rangle]\!] = \\
(\ \ Lift := \emptyset;\ \ ClosType := \emptyset;\ \ Dispatch := \emptyset; \\
\quad Lifted := \emptyset;\ \ ClosData := \emptyset; \\
\quad let\ \langle atdec'\rangle = Flatten\ \langle \mathcal{A}\,[\![atdecs]\!]\rangle\ \ in \\
\quad let\ \langle topdecs'\rangle = \mathcal{TS}_{\emptyset}\,[\![\langle topdecs\rangle]\!]\ \ in \\
\quad (^*\ at\ this\ point\ all\ mappings\ have\ been\ built\ ^*) \\
\quad closure\_atdecs := \emptyset;\ \ dispatch\_funs := \ \emptyset; \\
\quad for\ all\ tc \in Codom(ClosType)\ do \\
\qquad\quad closure\_atdecs := closure\_atdecs\ + \\
\qquad\qquad tc = \left\langle dc\ \texttt{of}\ \langle\tau\rangle_{*}\right\rangle_{|}; \\
\qquad\quad dispatch\_funs := dispatch\_funs\ + \\
\qquad\qquad Dispatch[tc]\!]\ \ (v_0 : tc\,,\langle v : \mu\rangle_{,})\ \ :\ \mu_0\ = \\
\qquad\qquad\qquad\qquad \texttt{case}\ v_0\ \texttt{of} \\
\qquad\qquad\qquad\qquad\quad \left\langle dc(\langle fv : \tau\rangle_{,})\ \texttt{=>}\ f(\langle v\rangle_{,}, \langle fv\rangle_{,})\right\rangle_{|} \\
\qquad\quad where\ \langle (dc, f, \langle fv : \tau\rangle)\rangle = ClosData(tc) \\
\qquad\quad and\ \langle\mu\rangle_{*} \rightarrow \mu_0 = ClosType^{-1}(tc) \\
\qquad\quad and\ v_0, \langle v\rangle\ are\ fresh \\
\quad done; \\
\quad \texttt{export}\ \langle export\rangle \\
\quad \texttt{datatype}\ \langle atdec'\rangle_{\texttt{and}}\ \texttt{and}\ \langle a \mid a \in closure\_atdecs\rangle_{\texttt{and}} \\
\quad \texttt{fun}\ \langle f \mid f \in dispatch\_funs\rangle_{\texttt{and}}\ \texttt{and}\ \langle f \mid f \in Lifted\rangle_{\texttt{and}} \\
\quad \langle topdecs'\rangle \\
)
\end{array}
$$

Fig. 19. Closure conversion of SIL components. The notation $\langle s \mid s \in S\rangle$ should be read as a sequence comprehension, i.e., the sequence of s values drawn from set S. Auxiliary function *Flatten* converts a sequence of sequences into a single sequence.

$\mathcal{FV}$ returns a set; we assume that an implementation will produce the members of the set in some deterministic order, which then becomes the canonical *sequence* ordering for the free variables wherever they are used. The free variable calculation is slightly tricky because it must produce the free variables of the *translated* term, given the original term as input; yet, the free variables must be calculated before the translation can occur! To break the circularity, we observe that the free variables sets of source and translated terms can only differ due to the replacement of a known function application $f(\langle v\rangle_{,})$ by the corresponding lifted application $Lift[f]\!](\langle v\rangle_{,}, \langle fv\rangle_{,})$, where $\langle fv\rangle$ are the free variables of f. In this case the target free variable set should not include f, but should include the $\langle fv\rangle$.[11]

The top-level conversion function $\mathcal{C}$ for components is shown in Figure 19. This function must be read imperatively, since the construction of the closure datatypes and dispatch functions and the translation of the export list rely on the auxiliary data structures built as a side-effect of the $\mathcal{TS}$ and $\mathcal{AS}$ translations. A datatype declaration and dispatch function are built for each closure datatype invented by *ClosType*, i.e., corresponding to each arrow type in the source program. Note that it is possible for a closure datatype to end up with *no* constructors; the corresponding

[11] We discovered this formulation of the free variable calculation in Xavier Leroy's Gallium compiler (Leroy, 1992).

dispatch function body is a case with no arms and hence no well-defined type. These dispatch functions are never actually applied; in most cases, the dead-code eliminator will remove them.

Freshly-created closure datatype declarations may refer to the converted versions of source program datatype declarations (since free variables may belong to datatypes) and vice-versa (since source datatypes may include fields of arrow type, which are converted to closure types). Therefore, the converted component has a *single* mutually recursive set of algebraic type declarations including both closure datatypes and converted source datatypes. For similar reasons, the converted component groups all the freshly-created closure dispatch functions and the lifted versions of the source program functions into a single mutually recursive declaration, followed by the translations of the original top-level declarations. Identifier uniqueness guarantees that it is harmless to declare any set of declarations as mutually recursive; a post-processing step is used to separate both datatypes and functions into their true mutually-recursive components.

9.3 Discussion

Because of the need for per-type dispatch functions, our algorithm depends critically on having monomorphic source code, but we believe a similar algorithm could be given for polymorphic programs with the addition of a typecase construct (Morrisett, 1995). Bell, Bellegarde and Hook (Bell et al., 1997) have specified a more elaborate algorithm for polymorphic source programs that performs type specialization and higher-order removal simultaneously, and may leave parts of the program polymorphic where that is possible. Their approach is thus more powerful, but it is also significantly more complicated, and has not been implemented.

Our algorithm also depends on having the full source program available; this restriction can be lifted if we permit *extensible* datatype declarations, i.e., datatypes for which the data constructor declarations can be scattered throughout the program, even in separate compilation units (Tolmach, 1997). Supporting such datatypes requires only a small extension to the type system (Standard ML treats the built-in exception type constructor in this way), but requires a somewhat more expensive implementation of case, and precludes the optimizations discussed in the next section.

10 Optimization of First-order Code

After first-order conversion and a pass back through the optimizer, a typical call to an unknown function has become a known call (to a dispatch function) followed by a case dispatch. This sequence is probably less efficient than the single indirect jump that would be performed by a conventionally closure-converted program.[12]

[12] In C, which supports indirect jumps to top-level functions, our representation could be converted back to a conventional closure representation as a final compilation step, by using the code pointers of the lifted functions as the constructor tags for the closure

However, there are many potential performance *advantages* to be obtained from the "interpreted" style of the converted program, deriving from the fact that it *is* an explicitly first-order program.

10.1 Uncurrying

The general-purpose optimizations that inline "small" functions and perform "case splitting" also work together on the explicit closure form to mimic the effect of a standard *uncurrying* transformation, with no extra implementation effort. Consider a curried function

```
f (x1:t1) (x2:t2) : t = e
```

expressed in SIL as:

```
fun f (x1:t1) : t2 -> t =
 let fun f2 (x2 : t2) : t  = e
 in f2
```

A fully-applied instance `((f` e_1`)` e_2`)` is expressed in SIL as:

```
let val g1 : t2 -> t = f e1
in g1 e2
```

This code is much less efficient than an application of an arity-2 function would be, due to the cost of building and entering an intermediate closure. An uncurrying transformation reduces the cost by introducing an arity-2 function `f'` and redefining `f` to call `f'` (note that e is not duplicated).

```
fun f' (x1:t1,x2:t2) : t = e
fun f (x1:t1) : t2->t =
  let fun f2 (x2:t2) : t = f'(x1:t1,x2:t2)
  in f2
```

Now fully-applied instances of `f` are altered to call `f'` directly instead; partially-applied or escaping instances of `f` are not changed. A similar transformation is desirable for curried functions of more than two arguments, whenever they are called with two or more actuals.

Uncurrying is ordinarily performed prior to closure conversion. Appel (1992) noted that uncurrying can be achieved simply by introducing the definition of `f'`, as above, and relying on standard inlining heuristics to inline `f` and `f2` (whose bodies are small), yielding a direct call to `f'`. Our observation is that *closure conversion* already performs the same transformation that Appel suggests, introducing a lifted version of `f2`. By applying a round of our standard optimizations *after* closure conversion, we get uncurrying "for free." Here is the result of closure conversion on the example above:

type. (This works because each closure value is cased over only once, by the relevant dispatch function.) Of course, the C code would require unsafe casts.

```
datatype clos = Cf2 of t1 | ...

fun f' (x1:t1) : clos = Cf2(x1)

fun dispatch (c:clos,x2:t2) : t =
   case c of
     Cf2 x1 => f2'(x2,x1)
   | ...

and f2' (x2:t2,x1:t1) : t  = e

let val g1 : clos = f'(e1)
in dispatch(g1,e2)
```

Now, the standard optimizer proceeds as follows: it inlines the call `f'`(e_1), since the body of the function is "small," which yields:

```
let val g1 : clos = Cf2(e1)
in dispatch(g1,e2)
```

Now the call to `dispatch` can be "case split," resulting in the inlining of the dispatch and yielding the direct n-ary call `f2'`(e_2, e_1)! Note that the success of this inlining strategy doesn't depend on the number of cases in this dispatch function, which might be arbitrarily large. Nor does it depend on a sizing heuristic; even our conservative inliner will *always* judge the relevant function bodies to be small enough. It also works correctly for functions of more than two arguments.

10.2 Implicit Type-based Closure Analysis

Higher-order functions 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 lambda-expressions that might be invoked at each application site in the program. Existing implementations of this so-called *closure analysis* use abstract interpretation involving a fixpoint calculation (Sestoft, 1988; Shivers, 1991), a constraint-based mechanism (Bondorf and Jørgensen, 1993; Palsberg, 1995), or region inference (Koch and Olesen, 1996). Surprisingly, no existing system appears to take advantage of the fact that simple typing provides a good first cut at the analysis "for free." Also, existing closure analysis algorithms do not express their results within the language itself, and so cannot feed subsequent general-purpose optimizations.

Our closure conversion algorithm can be seen as the encoding of a simple *type-based* closure analysis. Type inference tags each application site with a type, and the only lambda-expressions that can be invoked at that site are those whose type matches the annotation. The set of such lambdas is made explicit in the dispatch function called at that site and in the corresponding closure datatype. Standard partial-evaluation style optimizations such as constant propagation and dead code

```
datatype pclos flat = Ccons | Cconsapp of transform

fun foldl0 (n:transform,l:tlist) : transform =
  case l of
    TNil => n
  | TCons(x,r) =>
        let val n' : transform = compose(x,n)
        in foldl0(n',r)

fun foldl1 (c:pclos,n:plist,l:plist) : plist =
  case l of
    PNil => n
  | PCons(x,r) =>
        case c of
          Ccons =>
              let val n' : plist = Cons(x,n)
              in foldl1(c,n',r)
        | Cconsapp(whole_t) =>
              let val p0 : point = apply(whole_t,x)
              in let val n' : plist = Cons(p0,n)
                 in foldl1(c,n',r)

val ts : tlist  = TCons(...)

fun doit (ps:plist) : plist =
  let val whole_t : transform = foldl0(id,ts)
  in let val consapp : pclos = Cconsapp(whole_t)
     in let val ps0 : plist = foldl1(consapp,PNil,ps)
      in foldl1(Ccons,PNil,ps0)
```

Fig. 20. Optimized first-order code for geometric example component.

elimination, as described in Section 8, work directly on this representation. 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 knows precisely which function will be called, and can arrange to call it directly or (if it small enough) inline it (Jagannathan and Wright, 1996). Inlining is also possible (with some risk of code blow-up) for sets with just a few constructors, although we have not implemented this.

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 (Cardelli, 1984; Appel, 1992) will avoid building heap records for closure constructors with no free variables. It is also useful to support "flat" (i.e., unboxed) variant types (see Section 12.2) to avoid heap allocation for non-recursive constructors that have just a few free variables.

As an example, Figure 20 shows the effect of optimizing the code in Figure 15. Function `pdispatch`, having already absorbed `consapp'` and `cons'`, has been inlined into `foldl1`. Closure datatype `pclos` can be represented "flat" and hence need not be heap-allocated. Datatype `tclosure` has been recognized as a "unit" type, and its definition and uses have been removed altogether, allowing the body of `tdispatch` to be simplified into a call to `comp'` and thence into a call to `compose`, before being inlined into `foldl0`. Finally, function `doit'` has been inlined into `doit`.

The payoff from inlining and closure representation 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 programmer's choice of data structures and algorithms. However, source languages that support a name-equivalence model for types allow programmers to distinguish between different uses of structurally equivalent types. In RML (as in Standard ML), for example, this can be done by using "transparent" `datatype` declarations, e.g.,

```
datatype fahrenheit = F of int
datatype centigrade = C of int
```

Ordinarily, programmers do this in order to make their program text clearer and to obtain help from the compiler's type-checker in detecting logical errors. For example, lambda-bound functions of type `fahrenheit->fahrenheit` can be reliably distinguished from those of type `centigrade->centigrade`, etc., reducing the risk of accidentally confusing the two kinds of quantities. Under our closure conversion 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!

10.3 Explicit Closure Analysis

The translator can also perform its own forms of flow analysis explicitly, and record the results in the form of a more specialized typing, which the closure converter will take into account when collecting constructors into closure datatypes, thus perhaps producing a larger number of datatypes each containing fewer constructors. We have built one such analyzer, structured as a variant of type inferencing. Beginning with a copy of the original SIL program in which every expression is annotated with an explicit (monomorphic) type, the analyzer tags each occurrence of an arrow type (on a `fn` expression or a variable) 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 `fn` expression $(l : \tau_1 \rightarrow^i \tau_2)$ is among those 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 a refinement on ordinary types. This analysis is simple, given that the typed intermediate form in already in hand, and it is almost linear (its complexity is dominated by the union-find algorithm). It produces essentially the same analysis as the constraint-based approach described by Bondorf and Jørgensen (1993) and further analyzed by Palsberg (1995). Koch and Olesen (1996) have implemented a closure analysis for the ML Kit compiler based on (potentially polymorphic) region annotations (Tofte and Talpin, 1997); functions allocated to the same region are placed in the same closure-analysis equivalence class. Our tag unification method closely resembles region inference for monomorphic programs (Baker, 1990; Tofte and Talpin, 1997), although we developed it independently; nothing in our scheme corresponds to the ML Kit's polymorphic region inference, however.

An important point about our framework is that the *result* of an automated analysis like this can be expressed directly in SIL, and used as the basis of a (finer-grained) closure conversion. This is done by rewriting the SIL program. For each equivalence class $\tau_1 \rightarrow^i \tau_2$, the analyzer simply invents a new unary datatype `D`^{i} = `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 `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 transparent `D`^{i} types are cleaned from the resulting first-order program by the standard optimizer.

As future work, we plan to apply conventional (FORTRAN-world) optimizers to our closure-converted code, particularly to take advantage of well-developed dataflow frameworks that don't rely on inlining to propagate information.

11 Eliminating Tail Calls

Function calls are generally expensive in standard implementations of our target 3GLs.[13] So it is very desirable to remove tail calls in favor of jumps, especially when such calls are recursive. Tail calls are frequent in SIL, both in user functions derived from the original RML code, and in the `dispatch` functions generated by higher-order function removal.

To make it possible to express calls as jumps, SIL includes a facility for defining labeled *jump points* and corresponding *gotos* within a function (Kelsey, 1995). Jump points are declared similarly to `let`-bound functions, with a label name, formal parameters, defining expression, and body; `goto`s are similar to function applications, with a target jump point label and actual parameters. However:

- `goto` expressions can only appear in tail position;
- jump point labels can only be mentioned as the targets of `goto`s (i.e., they are not first class values); and
- a `goto` to a particular jump point may appear recursively inside the jump point definition, or within the immediate body, but *not* within any function declaration nested inside the body.

[13] Deeply recursive nests of calls are particularly expensive on SPARC processors when register windows are used (as they are by most 3GL compilers).

```
fun foldl0 (n:transform,l:tlist) : transform =
  let label jp0 (n0:transform,l0:tlist) : transform =
    case l0 of
      TNil => n0
    | TCons(x,r) =>
        let val n' : transform = compose(x,n0)
        in goto jp0(n',r)
  in goto jp0(n,l)

...

fun doit(ps:plist) : plist =
  let val whole_t = foldl0(id,ts)
  in ...
```

Fig. 21. Insertion of jump points into geometric example code.

Hence, when SIL is translated into a target 3GL, SIL jump point labels can become ordinary labels, their parameters become ordinary variable declarations scoped at the function level, and a SIL `goto` translates into a set of assignments to the parameter variables followed by an ordinary local `goto`.

When can a tail-call be turned into a `goto`? A simple recursive tail call from a function to itself is easy: a jump point is inserted at the top of the function body and the recursive tail call is changed to a `goto`; non-recursive and non-tail calls are unchanged. As an example, Figure 21 shows how a jump point is introduced at the top of `foldl0` in the code of Figure 20.

The same approach can be extended to handle tail-calls among mutually-recursive functions, though at a significantly increased cost. In order to make the nested labels have the proper scoping, the functions must be combined into a single function with simulated multiple entry points. A jump point is established inside the combined functions for each of the original functions, and the combined function gets an extra discriminant argument used to dispatch control to the appropriate label. The discriminant is encoded as a datatype, in a manner very similar to the closure datatypes introduced during higher-order function removal. For example:

```
let fun f (x:t1) = ... g z
    and g (y:t2) = ... f w ... f q
in g r
```

becomes

```
datatype D flat = F of t1 | G of t2
let fun f_or_g (d:D) =
  let label f (x:t1) = ... goto g z
      and   g (y:t2) = ... f_or_g(F w) ... goto f q
  in case d of
       F x' => goto f x'
     | G y' => goto g y'
in f_or_g(G r)
```

Under this transformation, a non-tail call to one of the original functions requires constructing a discriminant datatype value, passing it to the combined function, and performing an immediate case dispatch on it. Fortunately, the added datatype can always sensibly be declared `flat`, since it cannot be recursive, and its values are always consumed immediately at the top of the combined function and never escape. In most target 3GL compilers, the net effect is to push the datatype tag and parameters (i.e., the original functions' arguments) on the stack. In principle, good compilers could pass them in registers. Still, this transformation is costly in code size and execution time (for non-tail calls), so it is performed only if there is at least one tail-recursive call in the set of definitions. But it is well worth including in our repertoire, because mutual tail-recursion between `dispatch` functions and the lifted functions they invoke is quite common.

12 Generating C or Ada Code

The translation of first-order, optimized SIL code into our target 3GLs is fairly straightforward, so we give only a brief overview here. To ease re-targeting to new 3GLs, the translation is mediated by a common imperative intermediate form called MIL. A MIL component is a sequence of algebraic type declarations, value definitions, and function definitions. The translations to MIL and thence to Ada or C maintain the top-level structure of the SIL code.

12.1 MIL Code

The body of each MIL function is a *block*, which consists of local (mutable) variable declarations, a labeled set of nested sub-blocks, and an imperative statement sequence. A statement sequence consists of zero or more assignment statements (described below) followed by one of: an (unlabeled) sub-block, a `case` statement (whose arms are blocks), a `goto` to a label (in this block or an enclosing one), or an explicit function `return`. MIL blocks are produced for each SIL function body, `let`-binding, and `case` arm, and at several other points where it is convenient to use temporary variables. Each MIL block is translated directly to a {}-delimited block in C or a `DECLARE...BEGIN...END` block in Ada.

Assignment statements update variables declared in their block or in an enclosing one. The right-hand side of an assignment is restricted to be a variable, a literal constant, or an application; applications apply a primitive operator, constructor, or user-defined function to variables or constants. In particular, template-defined primitive operators appear only on the right-hand sides of assignments, and only with simple variable or constant arguments. MIL assignments are translated directly into C or Ada assignments.

12.2 Algebraic Type Representation

MIL algebraic type declarations extend SIL declarations with representation information. Careful choice of representations is quite important for achieving good performance in target code. Any algebraic type can be represented as a heap-allocated ("boxed"), tagged variant record, with each n-ary data constructor in the type corresponding to a tagged variant with n fields, and such types can be defined in a straightforward manner in Ada and C. Under this approach, all type representations have uniform size (one word), which is important for systems with polymorphic target code. More efficient representations that maintain the uniform size requirement are well-known; our system uses all the standard tricks (Appel, 1992; Cardelli, 1984) except those that require casting. But since our system generates *monomorphic* code, we need not require that all types have uniform size. Any non-recursive type whose values occupy a sufficiently small space can be represented *flat* (or "unboxed"), i.e., manipulated directly by value rather than being heap-allocated and manipulated by reference.

The use of unboxed records carries both benefits and costs. The major benefit is reducing the use of the heap, with consequent reductions in allocation, garbage collection, and data access costs. On the other hand, unboxed records are more expensive to move around than boxed ones, as each move requires that the entire contents of the record be copied. Thus automatic use of the unboxed representation should be restricted to fairly small records; we make the threshold size a tunable parameter of the translator. Users can also force a datatype to be held unboxed by marking it as `flat` in the source program.

Unlike other functional language compilers known to us, our translator supports unboxed representations even for variant records. These are particularly useful for avoiding heap-allocation of small closures. Of course, unboxed values always occupy the space needed for the largest possible variant, and hence waste space (and copying time) for smaller variants, so it is again important that the largest variant not be *too* large.

Both Ada and ANSI C support manipulation of unboxed record values, though not as efficiently as we would like. One potential advantage of using unboxed values is that they need not, in principle, stored in memory at all; they can often be profitably spread over registers (at least on machines that have lots of registers). Unfortunately, our target 3GL compilers are generally reluctant to handle unboxed records this way; in particular, they insist on passing and returning unboxed records on the stack. We cannot improve on this without direct access to machine code. Even so, choosing unboxed representations offers measurable improvements in the performance of some benchmarks, as discussed in Section 13.

12.3 Code Generation

We have used the `gcc` compiler for ANSI C compilation and the Sun Ada compiler (versions 1.1 and 3.0) for Ada83 compilation. We rely on the 3GL compilers to do several important tasks, including register allocation and copy propagation, peephole optimization of jumps, and generation of good code for case statements. In practice, the two compilers we use vary considerably in the quality of their code, with `gcc` generally doing a better job, especially on copy propagation.

In a few cases, the semantics of the target language cause subtle performance

Table 1. *Benchmark results.*

		`life`	`fft`	`interpd`	`interpc`	`sieve`	`mess`
	lines	302	237	113	119	48	385
`smlnj`[a]	time[b]	2.0	5.8	2.3	3.0	11.2	13.5
`standard`[c d]	time[b]	0.8	2.7	2.0	4.7	8.3	4.6
	closure size[e]	3	1	1	1	1	9
`flow-flat`[c d]	time[b]	0.8	2.7	1.7	4.6	8.2	4.6
`flow-boxed`[c d]	time[b]	1.0	2.7	2.0	5.1	8.2	4.7
	closure size[e]	3	1	5	7	1	9
`flow-flat-nogc`[c]	time[b]	0.5	2.6	1.2	2.4	5.1	3.4
	heap[f]	2.4	0	6.6	29.9	26.9	13.3
`flow-boxed-nogc`[c]	time[b]	0.6	2.7	1.3	2.6	5.1	3.2
	heap[f]	3.9	0	8.5	34.9	26.9	16.6

[a] SML/NJ version 109.27 with `reducemore := 0` and `rounds := 0`.
[b] User+system time in seconds, on unloaded 133MHz Pentium with 80MB memory, under Linux version 2.0.27.
[c] Generated C code compiled under `gcc` version 2.7.2.1 with option `-O3`.
[d] Generated C code linked with Boehm-Demers-Weiser conservative collector version 4.11.
[e] Maximum closure size in words.
[f] Heap allocation in MB.

problems. For example, in Ada83 a local variable slated to contain a variant record must be initialized with a default value, even if it is immediately overwritten by an assignment; these initializations make function entry much more expensive than the simple stack pointer adjustment one might expect.

We have also had to deal with a number of complications arising from arbitrary limitations in the Sun Ada compiler. For example, there is a hard internal limit on the depth of syntactically nested blocks; this has required us to perform a transformation on MIL function bodies that lifts all nested blocks to the top of the function. Unfortunately, this transformation broadens the syntactic scope of local variables and thus substantially increases the stress on the Ada compiler's register allocator.

13 Performance Benchmarks

Simple benchmark results indicate that our compiler generates code that is quite competitive in quality with the well-established Standard ML of New Jersey compiler. We also measure the effects of using more refined closure analysis and of using unboxed closure representations. A summary of the benchmark results is given in Table 1. `life` is an implementation by Reade (1989) of Conway's Game of Life making makes heavy use of higher-order functions; the inner loop processes a list of pairs of integers which we mark as `flat`. `fft` is an implementation of the Fast Fourier Transform due to Xavier Leroy; it is based on a template that supports simple operations on arrays of reals. `interpd` and `interpc` are lambda-calculus interpreters evaluating the factorial function; the former is in direct style and the latter in continuation-passing style; they are taken from Bondorf (1990). `sieve` is a list-based version of the sieve of Eratosthenes. `mess` parses and reformats simple bit-based messages; the RML code was generated by our Message Specification Language application generator (Kieburtz et al., 1995).

Row `smlnj` represents the behavior of Standard ML of New Jersey. The other rows represent the behavior of our compiler generating C under a variety of compilation settings; the resulting C was then compiled `gcc` and (unless otherwise noted) linked with the Boehm-Demers-Weiser conservative garbage collector (Boehm and Weiser, 1988). Row `standard` represents the standard configuration of our compiler. In particular, flat (non-heap) datatype representations are used for all non-recursive closure types. Execution times for our compiler are within a small factor of those of SML/NJ, and substantially better in some cases. These figures should be considered only as a rough indication of comparable performance, however, since there are numerous differences between the two systems that make exact comparison difficult. For example, SML/NJ checks for overflow on integer arithmetic operations, whereas C does not—although profiling results indicate that this difference is irrelevant to these particular benchmarks. More seriously, the performance of each system is heavily influenced by the amount of physical memory allocated by its memory management system at start-up, but it is difficult to control this quantity to ensure a fair comparison.

`flow-flat` represents a configuration in which we invoke the more explicit closure analysis described in Section 10.3 and continue to use the (often larger) flat representations for all closure types; `flow-boxed` does the same analysis but uses boxed representations for all closure types. Comparing these figures indicates that the refined closure analysis is occasionally worthwhile (e.g., for `interpd`), but only in conjunction with the flat representation for closure types.

These comparisons of flat vs. boxed closure representations may be skewed by our use of the relatively slow Boehm-Demers-Weiser collector, which probably penalizes heavy heap allocation disproportionately more than a system with an efficient built-in allocator. To get better evidence that flat closures types are worthwhile, we linked the generated code for `flow-flat` and `flow-boxed` against a very low-overhead heap memory management implementation: allocation from a single large array and no garbage collection. The results are shown as `flow-flat-nogc` and `flow-boxed-nogc`. Even with very cheap heap management, and despite the fact that `gcc` doesn't generate particularly good code for handling flat structures, the substantially lower heap allocation requirements of the `flat` approach lead to measurable speed improvement. We conclude that flat closure allocation is worth further investigation as an optimization technique for functional language compilers.

14 Conclusions

Versions of the compiler described here have been in use within our overall translation system for over two years. It generates working Ada83 and ANSI C code with respectable performance relative to established functional language compilers, and an unimpeachable level of type safety. It has cheerfully handled RML input programs of up to 20,000 lines. Generated Ada components have been integrated into the US Air Force's Generic Command Center demonstration environment, thus meeting the specific goals of the project for which this work was originally undertaken.

More broadly, we believe that our template approach is a promising alternative to previous interoperability schemes for strongly-typed functional languages. We would like to perform more detailed comparisons between our work and existing non-functional "glue" languages like Tcl. We also plan to extend template definitions to include specifications of algebraic laws that capture important primitive-specific optimizations such as arithmetic constant folding.

We also believe that our simple approaches to handling polymorphism and higher-order functions may be generally useful for implementing type-preserving compilers. Like other researchers (Tarditi et al., 1996; Peyton Jones, 1996) we have found the ability to type-check intermediate representations invaluable in uncovering bugs in the course of compiler development. Moreover, the type systems required by our intermediate languages are considerably simpler than those needed by some other systems. We have also developed new uses for type information in late-stage optimization of programs, and we see further opportunities to apply more conventional optimization techniques on first-order SIL code. The ability to generate efficient monomorphic data representations and to avoid some heap-allocation of closures has an important impact on performance. The effects of polymorphic function cloning on code size need more thorough investigation, however.

The most significant restriction of our system is that it requires access to the entire RML program, because both the polymorphism removal and higher-order removal algorithms are "whole-program" transformations. However, we believe that this problem can be at least partly addressed by providing separately compiled components a digest of the relevant type and function information from the other components.

References

Appel, A. W. (1992). *Compiling with Continuations.* Cambridge University Press.

Appel, A. W. and Jim, T. (1989). Continuation-passing, closure-passing style. In *Sixteenth ACM Symp. on Principles of Programming Languages*, pages 293–302, New York. ACM Press.

Appel, A. W. and Jim, T. (1998). Shrinking lambda expressions in linear time. *Journal of Functional Programming.* (to appear).

Baker, H. G. (1990). Unify and conquer (garbage collection, updating, aliasing, ...) in functional languages. In *Proc. 1990 ACM Conference on Lisp and Functional Programming*, pages 218–226.

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

Bell, J. M., Bellegarde, F., and Hook, J. (1997). Type-driven defunctionalization. In *Proc. 2nd International Conference on Functional Programming*, pages 25–37.

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

Boehm, H.-J. and Weiser, M. (1988). Garbage collection in an uncooperative environment. *Software—Practice and Experience*, 18(9):807–20.

Bondorf, A. (1990). Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17(1–3):3–34.

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

Cardelli, L. (1984). Compiling a functional language. In *Proc. 1984 ACM Conference on Lisp and Functional Programming*, pages 208–217.

Cardelli, L. (1987). Basic polymorphic typechecking. *Science of Computer Programming*, 8:147–172.

Chailloux, E. (1992). An efficient way of compiling ML to C. In *Proc. ACM Workshop on ML and its Applications*, pages 37–51.

Chin, W.-N. and Darlington, J. (1996). A higher order removal method. *Lisp and Symbolic Computation*, 9(4):287–322.

Cridlig, R. (1992). An optimizing ML to C compiler. In *Proc. ACM Workshop on ML and its Applications*, pages 28–36.

Damas, L. (1984). *Type Assignment in Programming Languages*. PhD thesis, University of Edinburgh.

Flanagan, C., Sabry, A., Duba, B. F., and Felleisen, M. (1993). The essence of compiling with continuations. *SIGPLAN Notices*, 28(6):237–247. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation.*

Hindley, R. (1969). The principal type-scheme of an object in combinatory logic. *Trans. Amer. Math. Soc.*, 146:29–60.

Huelsbergen, L. (1996). A portable C interface for Standard ML of New Jersey. Available at `http://cm.bell-labs.com/who/lorenz/papers/smlnj-c.ps`.

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

Jones, M. P. (1993). Partial evaluation for dictionary-free overloading. Technical Report YALEU/DCS/RR-959, Yale University Dept. of Computer Science.

Kelsey, R. A. (1995). A correspondence between continuation passing style and static single assignment form. In *Proc. ACM SIGPLAN Workshop on Intermediate Representations*, pages 13–22.

Kieburtz, R. B., Bellegarde, F., Bell, J., Hook, J., Lewis, J., Oliva, D., Sheard, T., Walton, L., and Zhou, T. (1995). Calculating software generators from solution specifications. In *TAPSOFT'95*, volume 915 of *LNCS*, pages 546–560. Springer-Verlag.

Koch, M. and Olesen, T. H. (1996). Compiling a higher-order call-by-value functional programming language to a RISC using a stack of regions. Master's thesis, University of Copenhagen. DIKU Dissertation 96-10-5.

Kranz, D., Kelsey, R., Rees, J., Hudak, P., Philbin, J., and Adams, N. (1986). Orbit: An optimizing compiler for Scheme. *SIGPLAN Notices*, 21(7):219–233. *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction.*

Lawall, J. L. and Danvy, O. (1993). Separating stages in the continuation-passing style transformation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 124–136, Charleston, South Carolina.

Leroy, X. (1991). The ZINC experiment: an economical implementation of the ML language. Technical Report 117, INRIA.

Leroy, X. (1992). Unboxed object and polymorphic typing. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 177–188.

Leroy, X. (1997). *The Objective Caml System: Documentation and User's Manual, Version 1.07.* INRIA.

Milner, R. (1978). A theory of type polymorphism in programming. *J. Computer and System Sciences*, 17:348–375.

Milner, R., Tofte, M., Harper, R., and MacQueen, D. (1997). *The Definition of Standard ML (Revised).* MIT Press, Cambridge, MA.

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

Morrisett, G. (1995). *Compiling with Types.* PhD thesis, Carnegie Mellon University. Available as TR CMU-CS-95-226.

Morrisett, G., Felleisen, M., and Harper, R. (1995). Abstract models of memory management. In *FPCA '95 SIGPLAN-SIGARCH-WG2.8 Conference on Functional Programming Languages and Computer Architecture*, pages 66–77.

Palsberg, J. (1995). Closure analysis in constraint form. *ACM Transactions on Programming Languages and Systems*, 17(1):47–62.

Peyton Jones, S. L. (1992). Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, pages 127–202.

Peyton Jones, S. L. (1996). Compilation by transformation: A report from the trenches. In *European Symposium on Programming (ESOP'96)*, volume 1058 of *LNCS*, pages 18–40.

Peyton Jones, S. L., Hall, C., Hammond, K., Partain, W., and Wadler, P. (1993). The Glasgow Haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference, Keele.*

Peyton Jones, S. L., Nordin, T., and Reid, A. (1997). Green Card: A foreign language interface for Haskell. In *Proc. ACM SIGLAN Haskell Workshop.*

Reade, C. (1989). *Elements of Functional Programming.* Addison–Wesley.

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

Sestoft, P. (1988). Replacing function parameters by global variables. Master's thesis, University of Copenhagen. DIKU Master's thesis no. 254.

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

Steele, G. L. (1978). Rabbit: a compiler for Scheme. Technical Report AI-TR-474, MIT, Cambridge, MA.

Talpin, J.-P. and Jouvelot, P. (1992). Polymorphic type, region, and effect inference. *Journal of Functional Programming*, 2(3):245–272.

Tarditi, D. (1996). *Design and Implementation of Code Optimizations for a Type-Directed Compiler for Standard ML.* PhD thesis, Carnegie Mellon University. Technical Report CMU-CS-97-108.

Tarditi, D., Lee, P., and Acharya, A. (1992). No assembly required: Compiling Standard ML to C. *ACM Letters on Programming Languages and Systems*, 1(2):161–177.

Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, R., and Lee, P. (1996). TIL: A type-directed optimizing compiler for ML. In Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation, pages 181–192.

Tofte, M., Birkedal, L., Elsman, M., Hallenberg, N., Olesen, T. H., Sestoft, P., and Bertelsen, P. (1997). Programming with regions in the ML Kit. Technical Report DIKU-TR-97/12, University of Copenhagen Department of Computer Science (DIKU).

Tofte, M. and Talpin, J.-P. (1997). Region-based memory management. *Information and Computation*, 132(2):109–176.

Tolmach, A. (1997). Combining closure conversion with closure analysis using algebraic types. In *Workshop on Types in Compilation TIC97.* Boston College Computer Science Technical Report BCCS-97-03.

Volpano, D. and Kieburtz, R. B. (1985). Software templates. In *Proceedings of the Eighth International Conference on Software Engineering*, pages 55–60. IEEE Computer Society.

Volpano, D. and Kieburtz, R. B. (1989). The templates approach to software reuse. In Biggersstaff, T. J. and Perlis, A. J., editors, *Software Reusability*, pages 247–255. ACM Press.

Wadler, P. (1989). Theorems for free! In *Proceedings 4th Int. Conf. on Functional Programming Languages and Computer Architecture*, pages 347–359.

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

Wright, A. K. (1995). Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–356.