From ML to Ada(!?!): Strongly-typed Language Interoperability via Source Translation

Dino P. Oliva
Andrew Tolmach

Pacific Software Research Center

Department of Computer Science
Oregon Graduate Institute
20000 N.W. Walker Road P.O. Box 91000
Portland, OR 97291-1000
oliva@cse.ogi.edu

Department of Computer Science
Portland State University
P.O. Box 751
Portland, OR 97207-0751
apt@cs.pdx.edu

June 2, 1997

Abstract

We describe a system that supports source-level integration of ML-like functional language code with C or Ada83 code, by translating the functional code into type-correct, "vanilla" 3GL code. The system offers simple, efficient, type-safe interoperation between new functional code components and "legacy" 3GL components. The novel features of our translator include 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 the result of a closure analysis.

This work was supported, in part, by the US Materiel Command under contract F19628-93-C-0069. The second author was also supported, in part, by the National Science Foundation under grant CCR-9503383.

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, defining and iterating over a list can be done more abstractly and succinctly in ML than in C. These high-level features 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 [17, 25], 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 (cast-free), portable, "vanilla" Ada83 or ANSI C code, which can be passed to a standard compiler. Since the output of the translator represents FL types and control structures using the 3GL's types and control structures, it can be easily integrated at a statement-by-statement level with the code of existing 3GL components, 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 [21]; 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

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

language— makes special demands on the translator. These have led us to introduce a number of novel techniques, some of which are of independent interest:

- We use a type-based macro-expansion technique called *templates* [42, 43] to integrate 3GL code into RML. RML programs are parameterized by a set of abstract types and operations, whose translations into target-language text are specified by macros. Both substantial legacy code components and simple primitive types and operators are handled uniformly in this fashion. Templates are specified using a specialized definition language; see Section 4.

- We remove 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 [19], and similar effects have sometimes been achieved by accident [41], we are unaware of any previous full-scale implementation. The approach requires access to the whole program.

- We remove higher-order functions using a novel 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 [27], we do not need to introduce new language primitives or fancy type systems to maintain typability, although we again require access to the whole program, which must be monomorphic.

- We *optimize* 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.* We take advantage of this closure analyses to choose more efficient closure representations and perform more aggressive inlining than an untyped conversion could support. We also show how to express the results of the somewhat stronger closure analysis of Bondorf and Jørgensen [9, 30] within the standard algebraic type framework.

- We eliminate tail-recursive calls, even among mutually recursive functions, without introducing global labels (which both ANSI C and Ada83 lack). We use 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 [1, 24, 32, 41]. 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. We rely 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 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 conservative collector [7]. 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 [33, 28], but in most cases types are abandoned well before code generation. The TIL compiler [41] 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 [40, 14, 12, 31], they treat C as a loosely-typed "portable assembly language," and often make use of casts and non-standard extensions (e.g., as provided by `gcc`).

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.

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'll also use Ada to write the "main program" or *driver* that will be responsible for invoking the RML component; we'll have little more to say about this driver, however.) In this application the granularity of primitive operations is quite small, so a function-based interface would not be very attractive. A template definition that imports these operations (and basic real number and boolean support)

[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 GeoLib IS
  TYPE trans_array IS ARRAY (integer RANGE 1..3, integer RANGE 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;
```

Figure 1: Example Ada Package Specification (Excerpts)

```
PACKAGE BODY GeoLib IS
  ...
  FUNCTION rotate (r:float) RETURN transform IS
    ret_val:transform; sinr:real := sin (r); cosr:real := cos (r);
  BEGIN
    ret_val := NEW trans_array'((cosr,-sinr,0.0),(sinr,cosr,0.0),(0.0,0.0,1.0));
    RETURN (ret_val);
  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;
```

Figure 2: Example Ada Package Implementation (Excerpts)

```
template GeoLibTemplate

header "WITH GeoLib; USE Geolib; WITH Math; USE Math"

type real (8) "float"
type point(16) "point"
type transform(4) "transform"

datatype bool "bool" = false "F" | true "T"

val / (x0:real,x1:real) : (res:real) "‘res‘ := ‘x0‘ / ‘x1‘;"
...
val not (b:bool) : (res:bool) pure
                "IF ‘b‘ = T THEN ‘res‘ := F ELSE ‘res‘ := T END IF;"
val id : transform "id"
val translate (x:real,y:real) : (res:transform)  "‘res‘ := translate (‘x‘,‘y‘);"
...
val apply (t: transform,p:point) : (res:transform)
                "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"
```

Figure 3: Example Template for Geometric Operations (Excerpts)

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 to this example component repeatedly, to 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 Figures 5–8. This is genuine output, except that we have renamed the variables for better readability. The output code illustrates many of the key characteristics of our translation approach; 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 inner function `f` of the original polymorphic `foldl` function has been specialized into two monomorphic variants `f0` and `f1`, taking `transform` lists to `transforms` and

```
export type point "point"
       type transform "transform"
       type point list "PointList"
       val Nil  : point list "PointNil"
       val Cons : point * point list -> point list "PointCons"
       type transform list "TransList"
       val Nil  : transform list "TransNil"
       val Cons : transform * transform list -> transform list "TransformCons"
       val doit : point list -> point list "doit"

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

val rec foldl (* : ∀ 'a,'b. ('a * 'b -> 'b) * 'b -> 'a list -> 'b *) =
  fn (c,n) => fn l =>
      let val rec f : 'b * 'a list -> 'b =
                  fn (n,l) =>
                        case l of
                          Nil => n
                        | Cons (x,r) => f (c (x,n),r)
    in f (n,l)


val ts (* : transform list *) = Cons (translate (2.0,~2.0),
                                       Cons (scale (1.0,0.5),
                                         Cons (rotate(/(3.141592,2.0)), Nil)))

val reverse (* : ∀ 'a.'a list -> 'a list *) = foldl(Cons, Nil)

val rec doit (* : point list -> point list *) =
 fn ps =>
   let val whole_t (* : transform *) = foldl (compose,id) ts
   in let val consapp (* : point * point list -> point list *) =
                fn (x,l) => Cons(apply(whole_t,x), l)
      in reverse(foldl (consapp,Nil) ps)
```

Figure 4: RML Component using Geometric Template. Type annotations are added as comments to improve readability.

`point` lists to `point` lists, respectively. The two possible functional arguments to `f1`, namely `Cons` and `consapp`, are represented as members of a discriminated record `PxPL2PL_clos`. 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 `f1`. The primitive Ada code for `apply`, used within `consapp`, has been inlined, as specified in the template. Even stronger optimization has been applied to `f0`: since `compose` is the *only* argument that can be passed to it, no closure is required at all, and its body

```
WITH GeoLib; USE GeoLib; WITH Math; USE Math;
PACKAGE Geo_package IS
  TYPE PointList_item ;  TYPE PointList IS ACCESS PointList_item;
  TYPE PointList_item IS RECORD PointCons_0:point;  PointCons_1:PointList; END RECORD;
  FUNCTION PointCons (PointCons_0:point; PointCons_1:PointList) RETURN PointList;
  PointNil:PointList := NULL;

  TYPE TransList_item ;  TYPE TransList IS ACCESS TransList_item;
  TYPE TransList_item IS RECORD TransCons_0:transform; TransCons_1:TransList; END RECORD;
  FUNCTION TransCons (TransCons_0:transform; TransCons_1:TransList) RETURN TransList;
  TransNil:TransList := NULL;

  FUNCTION doit (ps:PointList) RETURN PointList;
END Geo_package;
```

Figure 5: Generated Ada Package Spec Corresponding to Example

is specialized to call the primitive Ada `compose` routine directly.

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 `f0` and `f1` have been converted to local jumps. The only major remaining optimizations to be performed by the Ada compiler are variable coalescing and jump-to-jump elimination.

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 Revised Standard ML (SML '97) [26], without nested patterns or many derived forms, but with the addition of true multi-argument functions and data constructors. In this paper, we use a human-readable but still somewhat abstract syntax for RML (Figure 9) and the other intermediate languages used in the translator. In this representation, all identifier names are assumed to be distinct. In practice, source code is fed to the RML translator using a more elaborate concrete syntax (actually a subset of 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 [11] 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.

RML's typing rules are largely standard, so we mention only distinctive points here. RML

```
PACKAGE BODY Geo_package IS
  WITH GeoLib; USE GeoLib; WITH Math; USE Math;

  TYPE PxPL2PL_clos_constructors IS (cons_variant,consapp_variant);
  TYPE PxPL2PL_clos (constructor:PxPL2PL_clos_constructor := cons_variant) IS
  RECORD CASE constructor IS
    WHEN cons_variant => NULL;
    WHEN consapp_variant => whole_t:transform;
  END CASE; END RECORD;
  cons:PxPL2PL_clos(cons_variant);

  FUNCTION PointsCons (p0:point; p1:PointList) RETURN PointList IS
  BEGIN
    return NEW PointCons_item'(PointCons_0 => p0, PointCons_1 => p1);
  END;
  FUNCTION TransCons (t0:transform; t1:TransList) RETURN TransList IS
  BEGIN
    return NEW TransCons_item'(TransCons_0 => t0, TransCons_1 => t1);
  END;

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

  FUNCTION f0 (n:transform; l:TransList) RETURN transform IS
    n0:transform; l0:TransList;
  BEGIN
    n0 := n; l0 := l;
    GOTO JumpPoint0;
    <<JumpPoint0>>
    IF l0 = NULL THEN
      RETURN n0;
    ELSE
      DECLARE
        x : transform; r: TransList;
      BEGIN
        x := l0.TransCons_0; r := l0.TransCons_1;
        DECLARE
          n : transform;
        BEGIN
          n := compose(x,n0);
          n0 := n; l0 := r;
          GOTO <<JumpPoint0>>;
        END;
      END;
    END IF;
  END f0;
```

Figure 6: Generated Ada Code Body Corresponding to Example (beginning).

```
FUNCTION f1 (n:PointList; l:PointList; c: PxPL2PL_clos) RETURN PointList IS
  n0:PointList; l0:PointList; c0:PxPL2PL_clos;
BEGIN
  n0 := n; l0 := l; c0 := c;
  goto JumpPoint1;
  <<JumpPoint1>>
  IF l0 = NULL THEN
    RETURN n0;
  ELSE
    DECLARE
      x : point; r: PointList;
    BEGIN
      x := l0.PointCons_0; r := l0.PointCons_1;
      CASE c.constructor IS
        WHEN cons_variant =>
          DECLARE
            n : PointList;
          BEGIN
            n := NEW PointList_item'(PointCons_0 => x, PointCons_1 => n0);
            n0 := n; l0 := r; c0 := c0;
            GOTO <<JumpPoint1>>;
          END;
        WHEN consapp_variant =>
          DECLARE
            whole_t : transform;
          BEGIN
            whole_t := c0.whole_t;
            DECLARE
              p0 : point;
            BEGIN
              p0.x := ((x.x * whole_t0(1,1)) + (x.y * whole_t0(1,2)) + whole_t0(1,3));
              p0.y := ((x.x * whole_t0(2,1)) + (x.y * whole_t0(2,2)) + whole_t0(2,3));
              DECLARE
                n : PointList;
              BEGIN
                n := NEW PointList_item'(PointCons_0 => p0,PointCons_1 => n0);
                n0 := n; l0 := r; c0 := c0;
                GOTO JumpPoint1;
              END;
            END;
          END;
      END CASE;
  END IF;
END f1;
```

Figure 7: Generated Ada Code Body Corresponding to Example (continued)

```
  FUNCTION doit (ps:PointList) RETURN PointList IS
    whole_t:transform;
  BEGIN
    whole_t := f0(id,ts);
    DECLARE
      c : PxPL2PL_clos;
    BEGIN
      c := (consapp_variant,whole_t);
      DECLARE
        ps0 : PointList;
      BEGIN
        ps0 := f1(PointNil,ps,c);
        DECLARE
          ps1 : PointList;
        BEGIN
          ps1 := f1(PointNil,ps0,cons);
          RETURN ps1;
        END;
      END;
    END;
  END doit;

BEGIN
  t0 := translate(2.0,-2.0);
  t1 := scale(1.0,0.5);
  tf := 3.141592 / 2.0;
  t2 := rotate(tf);
  ts0 := NEW TransList_item'(TransCons_0 => t2,TransCons_1 => TransNil);
  ts1 := NEW TransList_item'(TransCons_0 => t1,TransCons_1 => ts0);
  ts := NEW TransList_item'(TransCons_0 => t0,TransCons_1 => ts1);
END Geo_package;
```

Figure 8: Generated Ada Code Body Corresponding to Example (conclusion)

abstract syntax includes explicit type annotations on variable and constructor mentions 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 [46].

As in SML '97, recursive bindings must be explicit function abstractions and polymorphic recursion among functions *and* datatypes is prohibited.[3] Unlike in SML, there are no records or tuples *per se*, but these can be built as datatypes with a single constructor. Also, datatypes can be marked as "`[flat]`" meaning that they should be manipulated as a tuple of immediate values

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

(types)	$\tau ::=$	K	(primitive types)
	$\mid$	t	(type variables)
	$\mid$	$(\{\tau\}_*) \to \tau$	(function types)
	$\mid$	$(\{\tau\}_,)D$	(algebraic types)
(type schemes)	$\sigma ::=$	$[\forall\{t\}_, .]\tau$	
(expressions)	$e ::=$	$(k : K)$	(primitive constants)
	$\mid$	$(v : \tau)$	(variables)
	$\mid$	$e(\{e\}_,)$	(function applications)
	$\mid$	$(c : \tau)(\{e\}_,)$	(constructor applications)
	$\mid$	$p(\{e\}_,)$	(primitive applications)
	$\mid$	$\texttt{fn}\ [\texttt{inline}]\ \mathit{rule}$	(anonymous abstractions)
	$\mid$	$\texttt{let}\ \mathit{vdecs}\ \texttt{in}\ e$	(local declarations)
	$\mid$	$\texttt{case}\ e\ \texttt{of}\ \{(c : \tau)\ \mathit{rule}\}_{\mid}$	(destructuring)
(rules)	$\mathit{rule} ::=$	$(\{v : \tau\}_,)\ \texttt{=>}\ e$	
(declarations)	$\mathit{vdecs} ::=$	$\texttt{val rec}\ \{v : \sigma = \texttt{fn}\ [\texttt{inline}]\ \mathit{rule}\}_{\texttt{and}}$	(recursive function decls.)
	$\mid$	$\texttt{val}\ v : \sigma = e$	(value decls.)
(algebraic type decls.)	$\mathit{atdec} ::=$	$(\{t\}_,)D[\texttt{flat}] = \{c\ [\texttt{of}\ \{\tau\}_*]\}_{\mid}$	
(mutually recursive decls.)	$\mathit{atdecs} ::=$	$\texttt{datatype}\ \{\mathit{atdec}\}_{\texttt{and}}$	
(imports)	$\mathit{import} ::=$	$\texttt{type}\ K$	(primitive type)
	$\mid$	$\texttt{datatype}\ (\{t\}_,)D$	(algebraic type)
	$\mid$	$\texttt{val}\ p : \tau$	(value)
(exports)	$\mathit{export} ::=$	$\texttt{type}\ \tau\ \texttt{"}\mathit{name}\texttt{"}$	
	$\mid$	$\texttt{val}\ v : \tau\ \texttt{"}\mathit{name}\texttt{"}$	

Figure 9: RML Abstract Syntax. In this and other syntax descriptions, we use the notation $\{x\}_{sep}$ 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.

$$
\begin{array}{lrll}
\text{(primitive types)} & \mathit{primtyp} ::= & \texttt{type}\ K\ \texttt{(}\mathit{size}\texttt{)}\ \texttt{"}\mathit{string}\texttt{"} & \\
\text{(algebraic types)} & \mathit{algtyp} ::= & \texttt{datatype}\ \texttt{(}\{t\}_{,}\texttt{)}\ D\ [\texttt{"}\mathit{string}\texttt{"}]\ [\texttt{flat}]\ \texttt{=}\ \{c\ [\texttt{"}\mathit{string}\texttt{"}]\ [\texttt{of}\ \{\tau\}_{*}]\}_{|} & \\
\text{(primitive values)} & \mathit{value} ::= & \texttt{val}\ k : K\ \texttt{"}\mathit{string}\texttt{"} & \text{(primitive constants)} \\
 & | & \texttt{val}\ p\texttt{(}\{v : \tau\}\texttt{):(}v : \tau\texttt{)}\ [\texttt{pure}]\ \texttt{"}\mathit{string}\texttt{"} & \text{(primitive functions)} \\
 & & & \\
\text{(templates)} & t ::= & \texttt{template}\ \mathit{name}\ [\ \texttt{header}\ \texttt{"}\mathit{string}\texttt{"}\ ]\ \{\mathit{primtype}\}\ \{\mathit{algtyp}\}\ \{\mathit{value}\} &
\end{array}
$$

Figure 10: Template specification syntax. Types τ are as in RML.

rather than being heap-allocated; this is suitable for small records or simple sum types (such as `option`). As a pathological special case, data types may have zero constructors; a `case` over such a constructor has no arms and thus arbitrary type, and its dynamic semantics is to abort.

The semantics of RML declarations and expressions are straightforward, so we omit a formal presentation. Primitives may have side-effects, and so can be used to provide mutable references or arrays and I/O operations. Like user functions, primitive receive their parameters by value. As in SML, evaluation order is fixed left-to-right, and all conditional control flow is governed by `case` expressions. 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 (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; transformations must not alter this mapping.

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 visible to RML code; this information is used by the translator when parsing and typechecking RML components. The template also includes macro definitions for the 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 10. 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.

Templates are primarily used to define abstract primitive types, values, and operators, whose representation and implementation are specified in terms of the target 3GL. These typically include both general-purpose primitive types (e.g., `integer`, `string`, ...) and application-specific types (e.g., `transform` or `point`). Primitive types are introduced by `type` declarations, which give 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. In addition, the size (in bytes) of the type's concrete realization is specified, to allow the translator to calculate the size of algebraic types that include the abstract type as a field. All primitive types must be monomorphic.

Primitive values and operators are defined by `val` declarations, which specify their expansion into 3GL code. A value declaration specifies the (RML) type of the value and the corresponding 3GL syntax for it.[4] 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 backquotes (`). For example, the definition of the primitive division operator might be

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

An RML expression like `val a = / (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 (MIL; see Section 12) 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

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

names or constants, which prevents potential problems with multiple uses of a formal argument in the macro.

Operators on primitive types (e.g., addition on integers) can often be implemented using built-in operators of the 3GL, whereas application-specific types usually depend on non-trivial 3GL type definitions and library code, but the template definition makes no formal distinction between them. If an expansion *string* references a 3GL library (Ada package or C file), the header declarations required to bring that library into scope should be placed in a `header` clause in the template specification. (`GeoLib` and `Math` are such libraries in our example.) If desired, calls to small functions can be inlined by hand in the operator definition (e.g., `apply` in our example).[5] Operators that have no side effects can be marked as `pure`; the translator can apply more aggressive optimizations to expressions that involve only pure operators (see Section 8).

In addition to abstract types, templates may include algebraic datatype declarations, just as in RML. Monomorphic instances of these types may appear in the type signatures of primitive operators. This facility is essential because all conditional control flow in RML is achieved by performing `case` operations over values of algebraic type. For example, the type `bool` is defined as the algebraic sum type `true | false`. If abstract operators were unable to return algebraic types such as `bool`, it would be impossible to perform conditional computation on the basis of their results. (The alternative of providing abstract conditional operators doesn't work in call-by-value languages.) Template-defined algebraic types may also be used in RML components translated against the template. There is no provision for mutually recursive combinations of algebraic and abstract types.

It is sometimes convenient for 3GL code expansions of operators to reference algebraic type constructors by name (e.g., the code for `not` in our example). To make this possible, monomorphic `datatype` declarations may include 3GL translation strings for the type name (e.g., `bool`) and the constructors (e.g., `T` and `F`); if present, these will be used in the generated 3GL code. The template designer must take care not to use names that clash with existing identifiers in the 3GL environment (such as the predefined Ada type `BOOLEAN` with constants `TRUE` and `FALSE`).

5 Compiler Architecture and Representations

The compiler is structured as a pipeline operating on a series of specialized, typed intermediate representations; see Figure 11. 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 from which imports are permitted.

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

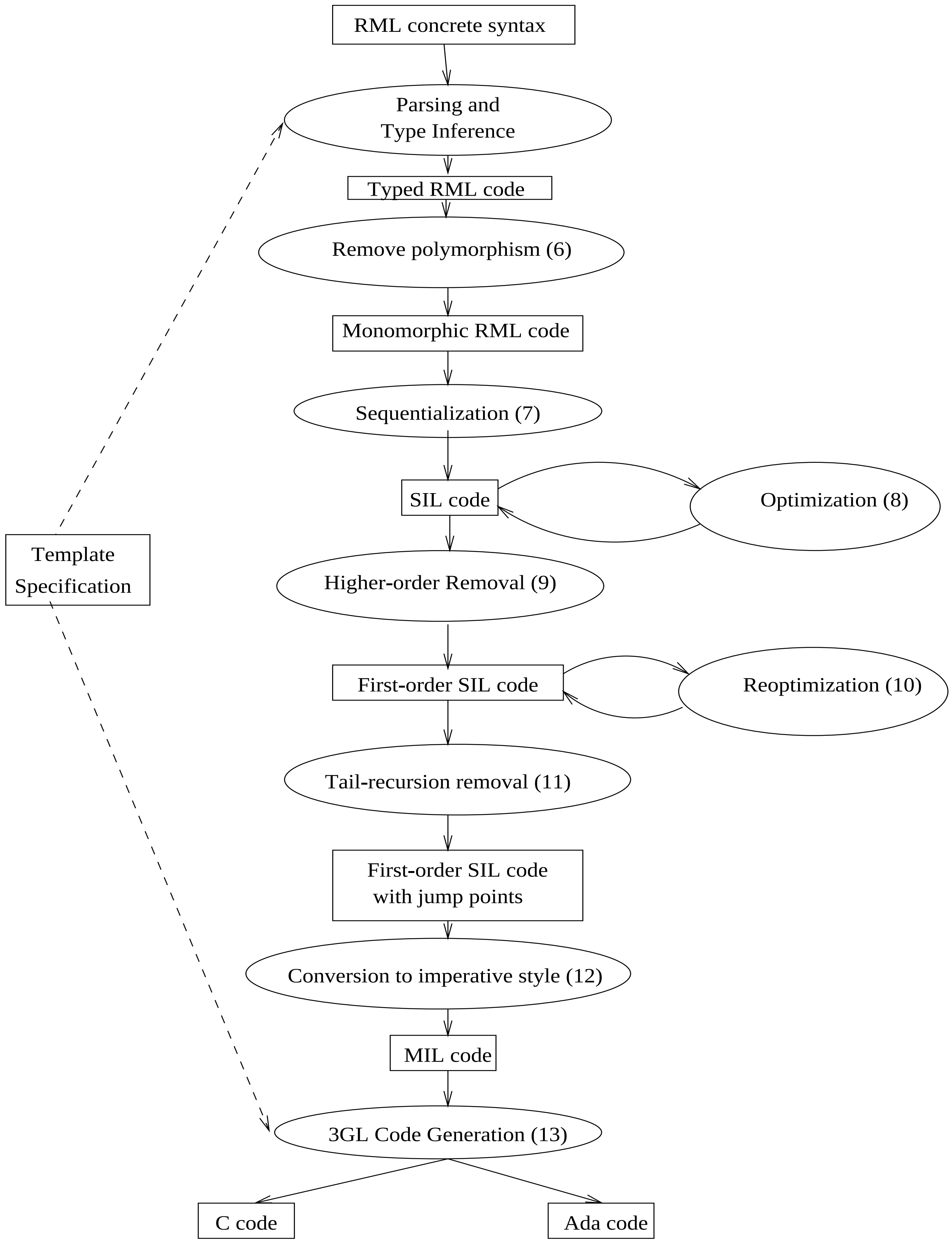

Figure 11: Architecture of the compiler. Numbers in parentheses refer to section numbers in this paper where the relevant operation is described.

- The RML code is annotated with type information using conventional Hindley-Milner type inference. The annotated code is then reduced 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 [16], closely related to CPS [38, 22, 1]). In SIL (Figure 15), 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 [20], though the initial translation to SIL doesn't use these.
- The SIL code is optimized (Section 9) 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 a final intermediate form, called MIL (for "Mutable Intermediate Language"), which abstracts the essential characteristics shared by C, Ada83, and similar languages (Section 12). MIL (Figure 28) is imperative; it has mutable variables and assignments rather than immutable values, statements rather than expressions, simple labels and `goto`s rather than jumps, and a variety of efficient representations for algebraic datatypes.
- MIL code is translated into Ada83 or C code using the template macros (Section 13).

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

6 Eliminating Polymorphism

6.1 Concept

Our target 3GLs do not directly support parametric polymorphism.[6] 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

[6] 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.

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 by recursively collecting the annotations at each level of nested polymorphism. 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 12.

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 [29]. 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 13. To explain the algorithm, we first require some terminology. A (simultaneous) *substitution* $S = (\{t\} \mapsto \{\tau\})$ is a mapping from a sequence of n type variables to a corresponding sequence of n types. Applying a substitution S to a type μ 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 $\{\mu\}$ to be the sequence $\{S(\mu)\}$. A *multi-substitution* M is a mapping $(\{t\} \mapsto \pmb{\{}\{\tau\}\pmb{\}})$ from a sequence 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. Here and elsewhere we use bold brackets ($\pmb{\{\}}$) to delimit sets, retaining ordinary brackets ($\{\}$ to denote syntactic sequences. 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 codomain of the mapping into a singleton set. If $M_1 = (\{t\} \mapsto T_1)$ and $M_2 = (\{t\} \mapsto T_2)$ are multi-substitutions with the same domain, we write $M_1 \uplus M_2$ for the multi-substitution $(\{t\} \mapsto T_1 \cup T_2)$, where $\cup$ represents ordinary set union with removal of duplicates. We define the *composition* $M_2 \circ M_1$ of multi-substitutions $M_1 = (\{t\} \mapsto T_1)$ and M_2 to be the multi-substitution $(\{t\} \mapsto \bigcup \pmb{\{}M_2(\{\tau\}) \mid \{\tau\} \in T_1\pmb{\}})$, where $\bigcup$ computes the union

```
export type point "point"
       type transform "transform"
       type list_point "PointList"
       val Nil_point  : list_point "PointNil"
       val Cons_point : point * list_point -> list_point "PointCons"
       type list_transform "TransList"
       val Nil_point  : list_transform "TransNil"
       val Cons_point : transform * list_transform -> list_transform "TransformCons"
       val doit : list_point -> list_point "doit"

datatype list_point = Cons_point of point * list_point | Nil_point
datatype list_transform = Cons_transform of transform * list_transform | Nil_transform

val rec foldl0 :
    (transform * transform -> transform) * transform -> list_transform -> transform =
             fn (c, n) =>  fn l =>
                     let val rec f : transform * list_transform -> transform =
                        fn (n,l) =>
                            case l of
                              Nil_transform => n
                                 | Cons_transform (x,r) => f (c (x,n),r)
                     in f (n,l)

val rec foldl1 :
    (point * list_point -> list_point) * list_point -> list_point -> list_point =
             fn (c, n) => fn l =>
                     let val rec f : list_poitn * list_point -> list_point =
                       fn (n,l) =>
                            case l of
                              Nil_point => n
                                 | Cons_point (x,r) => f (c (x,n),r)
                     in f (n,l)

val ts:list_transform = Cons_transform (...)

val reverse : list_point -> list_point =
      foldl1(fn (x:point,l:list_point) => Cons_point(x,l),Nil_point)

val rec doit : list_point -> list_point =
  fn ps =>
    let val whole_t : transform =
             foldl0 (fn (t1:transform, t2:transform) => compose(t1,t2),id) ts
    in let val consapp : point * list_point -> list_point =
                        fn (x,l) => Cons_point(apply(whole_t,x), l)
       in reverse(foldl1 (consapp,Nil) ps)
```

Figure 12: RML abstract syntax for example component after type specialization. Most type annotations are omitted for readability.

$$
\begin{aligned}
\mathcal{T}[\![K]\!] &= \emptyset \\
\mathcal{T}[\![(\{\tau\}_*) \to \tau]\!] &= (\biguplus\{\mathcal{T}[\![\tau]\!]\}) \uplus \mathcal{T}[\![\tau]\!] \\
\mathcal{T}[\![(\{\tau\}_,)\,D]\!] &= (\biguplus\{\mathcal{T}[\![\tau]\!]\}) \uplus \{D \mapsto (\mathit{Tyvarsof}[\![D]\!] \mapsto \{\tau\})\} \\
\\
\mathcal{S}[\![\forall\{t\}_,\,.\,\tau]\!] &= \mathcal{T}[\![\tau]\!] \\
\\
\mathcal{E}[\![(k : K)]\!] &= \emptyset \\
\mathcal{E}[\![(v : \tau)]\!] &= \{v \mapsto (\mathit{Inst}(\mathit{Schemeof}(v), \tau))\} \uplus \mathcal{T}[\![\tau]\!] \\
\mathcal{E}[\![e\,(\{e\}_,)]\!] &= \mathcal{E}[\![e]\!] \uplus (\biguplus\{\mathcal{E}[\![e]\!]\}) \\
\mathcal{E}[\![(c : \tau)\,(\{e\}_,)]\!] &= \{\mathit{Tyconof}[\![c]\!] \mapsto (\mathit{Inst}(\mathit{Schemeof}(c), \tau))\} \uplus \mathcal{T}[\![\tau]\!] \uplus (\biguplus\{\mathcal{E}[\![e]\!]\}) \\
\mathcal{E}[\![p\,(\{e\}_,)]\!] &= \biguplus\{\mathcal{E}[\![e]\!]\} \\
\mathcal{E}[\![\texttt{fn}\ \mathit{inl}\ \mathit{rule}]\!] &= \mathcal{R}[\![\mathit{rule}]\!] \\
\mathcal{E}[\![\texttt{let}\ \mathit{vdecs}\ \texttt{in}\ e]\!] &= \mathcal{D}(\mathcal{E}[\![e]\!])[\![\mathit{vdecs}]\!] \\
\mathcal{E}[\![\texttt{case}\ e\ \texttt{of}\ \{(c : \tau)\ \mathit{rule}\}_|]\!] &= \\
&\quad \mathcal{E}[\![e]\!] \uplus (\biguplus\{\{\mathit{Tyconof}[\![c]\!] \mapsto (\mathit{Inst}(\mathit{Schemeof}(c), \tau))\} \uplus \mathcal{T}[\![\tau]\!] \uplus \mathcal{R}[\![\mathit{rule}]\!]\}) \\
\\
\mathcal{R}[\![(\{v : \tau\}_,)\ \texttt{=>}\ e]\!] &= \{\mathcal{T}[\![\tau]\!] \uplus \mathcal{E}[\![e]\!]\} \\
\\
\mathcal{D}(I)[\![\texttt{val}\ v : \sigma\ \texttt{=}\ e]\!] &= I \uplus ((I[\![v]\!]) \circ (\mathcal{E}[\![e]\!] \uplus \mathcal{S}[\![\sigma]\!])) \\
\\
\mathcal{D}(I)[\![\texttt{val rec}\ \{v : \sigma\ \texttt{=}\ \texttt{fn}\ \mathit{inl}\ \mathit{rule}\}_{\texttt{and}}]\!] &= I \uplus ((\biguplus\{I[\![v]\!]\}) \circ (\biguplus\{\mathcal{R}[\![\mathit{rule}]\!] \uplus \mathcal{S}[\![\sigma]\!]\})) \\
\\
\mathcal{DS}(I)[\![\{\mathit{vdecs}\}\ \mathit{vdec}]\!] &= \mathcal{DS}(\mathcal{D}(I)[\![\mathit{vdec}]\!])[\![\{\mathit{vdecs}\}]\!] \\
\mathcal{DS}(I)[\![\ ]\!] &= I \\
\\
\mathcal{AS}(I)[\![\{\mathit{atdecs}\}\ \mathit{atdecs}]\!] &= \mathcal{AS}(\mathcal{A}(I)[\![\mathit{atdecs}]\!])[\![\{\mathit{atdecs}\}]\!] \\
\mathcal{AS}(I)[\![\ ]\!] &= I \\
\\
\mathcal{A}(I)[\![\texttt{datatype}\ \{(\{t\}_,)\,D\mathit{flt} = \{c\ [\texttt{of}\ \{\tau\}_*]\}_|\}_{\texttt{and}}]\!] &= \\
&\quad I \uplus (I[\![D]\!]) \circ (\biguplus\{(\biguplus\{(\biguplus\{\mathcal{T}[\![\tau]\!]\})\})\}) \\
\\
\mathcal{X}[\![\texttt{type}\ \tau\ \texttt{"}\mathit{name}\texttt{"}]\!] &= \mathcal{T}[\![\tau]\!] \\
\mathcal{X}[\![\texttt{val}\ v : \tau\ \texttt{"}\mathit{name}\texttt{"}]\!] &= \{v \mapsto \mathit{Inst}(\mathit{Schemeof}(v), \tau)\} \uplus \mathcal{T}[\![\tau]\!] \\
\\
\mathcal{M}[\![\texttt{export}\ \{\mathit{export}\}\{\mathit{atdecs}\}\{\mathit{vdecs}\}]\!] &= \mathcal{DS}(\mathcal{AS}(\biguplus\{\mathcal{X}[\![\mathit{export}]\!]\})[\![\{\mathit{atdecs}\}]\!])[\![\{\mathit{vdecs}\}]\!]
\end{aligned}
$$

Figure 13: Enumeration of instances of polymorphic identifiers.

of the members of a set of sets.

An *instantiation map* $I[\![x]\!]$ is a mapping from polymorphic identifiers $x : \forall\{t\},.\tau_0$ to multi-substitutions with domain $\{t\}$; we will build instantiation maps whose domains includes both variables and algebraic type constructors. 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 $\{I\}$ is a sequence of instantiation maps, $\biguplus\{I\}$, represents their sequential combination under $\uplus$. 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 13 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, $\mathcal{M}$ 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 relies on a number of auxiliary functions. *Inst(*$\forall\{t\}_,.\tau_0,\tau$*)* returns a substitution $S = (\{t\} \mapsto \{\mu\})$ such that $S(\tau_0) = \tau$; it will only be called on arguments for which such the result substitution is guaranteed to exist. Note that the μ may still contain other type variables. 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 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 $Tyvarsf$ on the type constructor(s). These conditions are naturally met by the annotations produced by a standard type-inferencer.

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 $\mathcal{A}$. Because RML prohibits polymorphic recursive definitions of functions or algebraic types, 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. Thus, for example, to process a definition fragment `let val` $v : \forall\{t\}_,.\tau$ `=` e_1 `in` e_2, the algorithm

i. builds an instantiation map (I) based on e_2;

ii. builds another instantiation map based on e_1, in which the instantiating types may mention the type variables $\{t\}$;

iii. expands this latter map by pre-composing with $I[\![v]\!]$, the multi-substitution describing all possible instantiations for the $\{t\}$;

```
datatype 'a List  = Nil | Cons of 'a * 'a List
val f : ∀ 'b.'b -> 'b List =
            fn (x: 'b) => ((Cons:'b * 'b List -> 'b List) ((x:'b), (Nil:'b List)))
val g : ∀ 'c,'d. 'c * 'd -> 'd List =
            fn (y:'c, z: 'd) =>
                let w : 'c List = (f: 'c - 'c List) (y:'c)
                in (f:'d -> 'd List) (z:'d)
val a : bool List = (g:int * bool -> bool List) (3:int,true:bool)
val b : string List = (g:int * string -> string List) (1:int,"abc":string)
```

Figure 14: Example of nested polymorphic functions.

iv. adds the expanded map to I to produce the complete map for the definition fragment.

The rules for recursive function and datatype definitions are similar. Note that for the recursive function case it is necessary to combine instance information about uses of *all* the functions before pre-composing.

As an (artificial) example, consider the code in Figure 14, written in explicitly typed form. The computation proceeds roughly as follows:

- The right-hand side for `b` is processed, yielding an instantiation map

$$I_1 = \{\mathtt{g} \mapsto (\{'c,'d\} \mapsto \{\{\mathtt{int}, \mathtt{string}\}\})\}$$

  which also serves as the map for the overall declaration of `b`

- The right-hand side for `a` is processed, yielding an instantiation map

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

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

$$I_3 = \{\mathtt{g} \mapsto (\{'c,'d\} \mapsto \{\{\mathtt{int}, \mathtt{string}\}, \{\mathtt{int}, \mathtt{bool}\}\})\}$$

- The right-hand side for `g` is processed, yielding (in several steps) the map

$$I_4 = \{\mathtt{f} \mapsto (\{'b\} \mapsto \{\{'c\}, \{'d\}\}), \mathtt{List} \mapsto (\{'a\} \mapsto \{\{'c\}, \{'d\}\})\}$$

- The composition $I_3(\mathtt{g}) o I_4$ is computed, yielding the map

$$I_5 = \left\{ \begin{array}{l} \mathtt{f} \mapsto (\{'b\} \mapsto \{\{\mathtt{int}\}, \{\mathtt{string}\}, \{\mathtt{bool}\}\}), \\ \mathtt{List} \mapsto (\{'a\} \mapsto \{\{\mathtt{int}\}, \{\mathtt{string}\}, \{\mathtt{bool}\}\}) \end{array} \right\}$$

- I_5 is added to I_3 to produce the overall map for the declarations of `g`, `a`, and `b`:

$$I_6 = \left\{ \begin{array}{l} \mathtt{g} \mapsto (\{'c,'d\} \mapsto \{\{\mathtt{int}, \mathtt{string}\}, \{\mathtt{int}, \mathtt{bool}\}\}), \\ \mathtt{f} \mapsto (\{'b\} \mapsto \{\{\mathtt{int}\}, \{\mathtt{string}\}, \{\mathtt{bool}\}\}), \\ \mathtt{List} \mapsto (\{'a\} \mapsto \{\{\mathtt{int}\}, \{\mathtt{string}\}, \{\mathtt{bool}\}\}) \end{array} \right\}$$

After possible further expansion by processing of `reverse`, I_6 can 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 `('c,'d)`; three specialized copies are made of `List` and `f`, corresponding to the three instantiations for `'a` and for `'b`. We omit a detailed description of this pass, which is quite straightforward given the existence of the instantiation map.

6.3 Discussion

In the worst case, the number of cloned versions of a function or datatype may be exponential in the static nesting depth of the 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 [44]!

The idea of removing parametric polymorphism by specialization has received much informal discussion, and a small experiment has been attempted for Gofer [19], but we are not aware of any previous full-scale implementation based on this approach. Analysis of benchmarks run on the Til compiler [41] 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 3GL target languages are 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 15. 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); otherwise, imports, exports, and algebraic type declarations are identical to RML. Jump points (`label` and `goto`) and their use are discussed in Section 11.

Figure 16 gives the details of the RML to SIL translation of expressions and declarations, using syntactic continuations. The symbol $\overline{\lambda}$ introduces a meta-level function, which is applied using $\overline{@}$; none of these meta-functions appear in the final output. The translation $\mathcal{E}[\![e]\!]$ of each expression e generates a SIL-language expression and a simple expression for it; the translation function is

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

Figure 15: SIL syntax.

$$
\begin{array}{rcl}
\mathcal{E}\llbracket(k:\tau)\rrbracket & = & \overline{\lambda}\kappa.\kappa\overline{@}(k:\tau) \\
\mathcal{E}\llbracket(v:\tau)\rrbracket & = & \overline{\lambda}\kappa.\kappa\overline{@}v \\
\mathcal{E}\llbracket e_0(\{e\}_,)\rrbracket & = & \overline{\lambda}\kappa.\mathcal{E}\llbracket e_0\rrbracket\overline{@}(\overline{\lambda}se.\mathcal{ES}\llbracket\{e\}_,\rrbracket\overline{@}(\overline{\lambda}\{se\}_,.\texttt{let val }v:\mu\texttt{ = }se(\{se\}_,)\texttt{ in }\kappa\overline{@}v)) \\
 & & \hfill (v \text{ fresh}; \mu = \mathit{Codom(Typeof(}e_0\mathit{)))} \\
\mathcal{E}\llbracket(c:\tau)(\{e\}_,)\rrbracket & = & \overline{\lambda}\kappa.\mathcal{ES}\llbracket\{e\}_,\rrbracket\overline{@}(\overline{\lambda}\{se\}_,.\texttt{let val }v:\mu\texttt{ = }c(\{se\}_,)\texttt{ in }\kappa\overline{@}v) \\
 & & \hfill (v \text{ fresh}; \mu = \mathit{Codom(}\tau\mathit{))} \\
\mathcal{E}\llbracket p(\{e\}_,)\rrbracket & = & \overline{\lambda}\kappa.\mathcal{ES}\llbracket\{e\}_,\rrbracket\overline{@}(\overline{\lambda}\{se\}_,.\texttt{let val }v:\mu\texttt{ = }p(\{se\}_,)\texttt{ in }\kappa\overline{@}v) \\
 & & \hfill (v \text{ fresh}; \mu = \mathit{Codom(Typeof(p)))} \\
\mathcal{E}\llbracket\texttt{fn }\mathit{inl}\ \mathit{rule}\rrbracket & = & \overline{\lambda}\kappa.\texttt{let fun }v\ \mathit{inl}\ \mathcal{R}_F\llbracket\mathit{rule}\rrbracket\texttt{ in }\kappa\overline{@}v \qquad (v \text{ fresh}) \\
\mathcal{E}\llbracket\texttt{let }\mathit{decs}\texttt{ in }e\rrbracket & = & \overline{\lambda}\kappa.\texttt{let }\mathcal{D}\llbracket\mathit{decs}\rrbracket\texttt{ in }\llbracket e\rrbracket\overline{@}\kappa \\
\mathcal{E}\llbracket\texttt{case }e_0\texttt{ of }\{(c:\tau)\ \mathit{rule}\}_|\rrbracket & = & \overline{\lambda}\kappa.\mathcal{E}\llbracket e_0\rrbracket\overline{@}(\overline{\lambda}se.\texttt{let val }v:\mu\texttt{ = case }se\texttt{ of }\{c\ \mathcal{R}_C\llbracket\mathit{rule}\rrbracket\}_|\texttt{ in }\kappa\overline{@}v) \\
 & & \hfill (v \text{ fresh}; \mu = \mathit{Codom(}\tau\mathit{))}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{ES}\llbracket e_0,\{e\}_,\rrbracket & = & \overline{\lambda}\kappa.\mathcal{E}\llbracket e_0\rrbracket\overline{@}(\overline{\lambda}se_0.\mathcal{ES}\llbracket\{e\}_,\rrbracket\overline{@}(\overline{\lambda}\{se\}_,.\kappa(se_0,\{se\}_,))) \\
\mathcal{ES}\llbracket\rrbracket & = & \overline{\lambda}\kappa.\kappa\overline{@}\{\}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{D}\llbracket\texttt{val }v:\tau\texttt{ = }e\rrbracket & = & \texttt{val }v\texttt{ = }\mathcal{E}\llbracket e\rrbracket\overline{@}(\overline{\lambda}se.se) \\
\mathcal{D}\llbracket\texttt{val rec }\{v:\sigma\texttt{ = fn }\mathit{inl}\ \mathit{rule}\}_{\texttt{and}}\rrbracket & = & \texttt{fun }\{v\ \mathit{inl}\ \ \mathcal{R}_F\llbracket\mathit{rule}\rrbracket\}_{\texttt{and}}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{R}_F\llbracket(\{v:\tau\}_,)\texttt{ => }e\rrbracket & = & (\{v:\tau\}_,)\texttt{ : }\tau\texttt{ = }\mathcal{E}\llbracket e\rrbracket\overline{@}(\overline{\lambda}se.se) \qquad (\tau = \mathit{Typeof(e)}) \\
\mathcal{R}_C\llbracket(\{v:\tau\}_,)\texttt{ => }e\rrbracket & = & (\{v:\}_,)\texttt{ => }\mathcal{E}\llbracket e\rrbracket\overline{@}(\overline{\lambda}se.se)
\end{array}
$$

Figure 16: Transformation from RML to SIL, specified using syntactic continuations. Auxiliary function *Typeof* reconstructs the type of an RML term, and *Codom* returns the co-domain of an arrow type.

parameterized by a continuation meta-function κ that says "what to do" with this simple expression. The translation of expression sequences ($\mathcal{ES}\llbracket\{e\}_,\rrbracket$) encodes RML's left-to-right evaluation order; it generates a sequence of names of the expressions and its continuation meta-function says "what to do" with the sequence of names. The translations for declarations ($\mathcal{D}$) and function or `case` rules ($\mathcal{R}$) are direct-style and straightforward. This style of translation algorithm, which works in one pass and generates no unnecessary fresh names, is due to Danvy and Filinski [15].

As an example, Figure 17 shows the SIL form of the `foldl0`, `reverse` and `doit` functions from the monomorphic version (Figure 12) of our running example from Section 2.

The semantic correctness of the transformation is straightforward to prove. The one significant detail to be checked is that left-to-right order of evaluation is correctly encoded in the translation rules, e.g., for $\mathcal{ES}$. Since the RML source is monomorphic type preservation is almost trivial.

The essentials of this transformation are well-known; they are very similar to a partial continuation-passing-style (CPS) transform [38, 22, 1]. Recently, there has been considerable interest in "almost-CPS" translations, which perform the naming and sequentialization steps of a CPS

```
fun foldl0 (c:transform * transform -> transform, n: transform) :
                                               list_transform -> transform =
  let fun g (l:list_transform) : transform =
    let fun f (n: transform, l:list_transform) : transform =
          case l of
            Nil_transform => n
          | Cons_transform(x,r) =>
                  let val n' : transform = c(x,n)
                  in f(n',r)
     in f(n,l)
  in g

fun foldl1 (c:point * list_point -> list_point, n: list_point) :
                                               list_point -> list_point =
   ... same as foldl0 except for types...

val reverse : list_point -> list_point =
  let fun cons (p:point, c:list_point) : list_point = Cons_point(p,c)
  in foldl1(cons,Nil_point)

fun doit (ps:list_point) : list_point =
  let val whole_t : transform  =
      let fun comp (x:transform,y:transform) : transform = compose (x,y)
      in let val f : list_transform -> transform  = foldl0 (comp,id)
         in f ts
  in let fun consapp (x:point,l:list_point) : list_point =
       let val x' : point  = apply (whole_t,x)
       in Cons_point (x',l)
     in let val f : list_point->list_point = foldl1 (consapp,Nil_point)
        in let val ps' : list_point = f ps
           in reverse ps'
```

Figure 17: Initial SIL translation of example program (selections).

transform, but don't introduce full-scale continuations [23, 16, 20, 39].

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,[7] elimination of unused function and (pure) value bindings, elimination of unused datatypes, and conservative function inlining. A function application is inlined if

- it is the sole application of that function; or

[7]Note that this includes as a special case the selection of fields from records with known values.

```
fun foldl1 (c:point * list_point -> list_point, n: list_point) :
                                                    list_point -> list_point =
  ... same as in Figure 17...

fun doit (ps:list_point) : list_point =
  let fun f0 (n: transform,l:list_transform) : transform =
          case l of
            Nil_transform => n
          | Cons_transform(x,r) =>
               let val n' = compose(x,n)
               in f0(n',r)
  in let val whole_t : transform = f0(id,ts)
     in let fun consapp (x:point,l:list_point) : list_point =
            let val x' : point  = apply (whole_t,x)
            in Cons_point (x',l)
        in let val f : list_point->list_point = foldl1 (consapp,Nil_point)
           in let val ps' : list_point = f ps
              in let fun cons (p:point, c:list_point) : list_point = Cons_point(p,c)
                 in let val g : list_point -> list_point = foldl1(cons,Nil_point)
                    in g ps'
```

Figure 18: Result of optimizing example program (selection).

- 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 [3]. We do not perform speculative inlining. Optimization passes are performed repeatedly until no change is observed or some fixed small number of passes has been reached. We precede these optimizing passes by a single round of eta-expansion to improve opportunities for inlining.

When the initial SIL translation (Figure 17) of our example program from Section 2 is optimized, function `foldl0`, which is used only once, is in-lined into the body of `doit` with arguments `c` and `n` specialized. Value `reverse` is eta-expanded into a function which is also inlined into `doit`. The resulting version of `doit` is shown in Figure 18. The body of function `foldl1` is unchanged by optimization.

Note that in the body of `doit`, the local function `f0` used to calculate `whole_t` has been hoisted to a more global scope. This is not essential; SIL permits the defining expression in a `let` binding to be another `let` binding or a `case` expression. This flexibility keeps SIL closed under the operation of function inlining; otherwise, it would be necessary to renormalize every inlined expression (e.g., as in [39]). However, though not required, hoisting `case` expressions out of `let`s can aid optimization by increasing the amount of information available for constant propagation in each case arm. The general form of the transformation is:

```
let val v = case e0 of                    becomes     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
```

In general, this is a dangerous transformation, since it duplicates the code for e in each case arm. However, it is worth doing if e has the form $f(v)$ and we can perform case splitting on f. Even if case splitting is not possible, code explosion will not a problem so long as e is a "small" expression, so we perform the transformation in just that those circumstances.[8] We always hoist `lets` out of `lets`, as this never hurts, and may help by exposing more `case` hoisting opportunities. Each of these hoisting transformations is done as a separate pass following the main simplification pass.

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 non-terminating 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 the number of eliminable expressions (e.g., [39]).

9 Removing Higher-order Functions

9.1 Concepts

Our target 3GL's 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'd like to express the first-order programs in a strict subset of the original language, as in "closure-passing style" [2], 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 in doing

[8]Code explosion could be avoided in the all cases by introducing a continuation function to be applied by each case arm. If the continuation function can itself be `case`-split, this may be worthwhile, but if it cannot, we end up having *added* function call overhead! Moreover, although these local continuation functions would be natural candidates for turning into (much cheaper) jump points (see Section 11), they will have been lifted out to top-level by the higher-order removal algorithm (see Section 9) before this can happen.

this is that two functions having the same argument and result 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 [27] 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. Moreoever, 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 [35] and Warren [45] and explored in typed settings by Bell, Bellegarde and Hook [4, 6, 5]. The key idea is to represent function closures as members of an algebraic data type (i.e., discriminated union). There is one constructor for each lambda expression in the program; its arguments are the lambda expression's free variables.[9] To convert a program to first order, a suitable closure type declaration is introduced, lambda expressions are transformed into closure constructor applications, and calls to "unknown" (i.e. lambda-bound) functions are transformed into calls to an auxiliary function that dispatches on the constructor to invoke a lambda-lifted version of the correct original function. 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 19 provides a simple example.

In a strongly-typed setting, we cannot make do with just one closure datatype and dispatch function: we must have a pair of them for each distinct arrow type in the program. The translation algorithm chooses the correct dispatch function at each site by inspecting the type of the (original) function. For example, if we alter the above example to use a continuation-passing version of `map`, as in Figure 20, we need two sets of `clos` datatypes and `dispatch` functions. Executing this example generates a statically unbounded number of distinct `list_int -> list_int` closures, one per member of `q`, each containing another such closure as a free variable. This is reflected in the fact that type `clos_li2li` is recursive. Note that higher-order removal techniques based on code specialization [13] cannot cope with programs of this sort.

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 lists with new arguments representing the function's free variables;

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

Original SIL code:

```
datatype list_int = Nil_int | Cons_int of int * list_int
fun outer (y:int,q:list_int) : list_int =
  let fun g1 (x:int) : int = + (x,y)
  in let fun g2 (x:int) : int = - (x,2)
     in let fun map (f:int->int,l:list_int) : list_int =
                  case l of
                    Nil_int => Nil_int
                  | Cons_int(h,t) =>
                             let val c1:int = f h
                         in let val c2:list_int = map (f,t)
                            in Cons_int(c1,c2)
        in map (g1, q)
```

After closure-converting g1 and g2:

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

fun outer (y:int,q:list_int) =
  let val g1:clos = G1 y               (* closures *)
  in let val g2:clos = G2
     in let fun map(f:clos,l:list_int) : list_int =
                  case l of
                    Nil_int => Nil_int
                  | Cons_int(h,t) =>
                          let val c1:int = dispatch(f,h)
                          in let val c2:list_int = map(f,t)
                             in Cons_int(c1,c2)
        in map (g1, q)
```

Figure 19: Simple example of typed closure conversion

- 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;[10]
- 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;

[10]Special *stub* versions of top-level functions are created to avoid changing the signatures of exported values; see below.

Original SIL code:

```
datatype list_int = Nil_int | Cons_int of int * list_int
fun outer (y:int, q:list_int) : list_int =
  let fun g1 (x:int) : int = + (x,y)
  in let fun g2 (x:int) : int = - (x,2)
     in let fun id (p:list_int) : list_int = p
        in let fun map (f:int->int,l:list_int,k:list_int -> list_int) : list_int =
                     case l of
                       Nil_int => k Nil_int
                     | Cons_int(h,t) =>
                            let fun kk(p:list_int) : list_int =
                                 let val c1 = f h
                                 in let val c2 = Cons_int(c1,p)
                                    in k c2
                          in map(f,t,kk)
           in map (g1,q,id)
```

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

```
datatype list_int = Nil_int | Cons_int of int * list_int
and clos_i2i = G1 of int | G2                          (* int -> int closure datatype *)
and clos_li2li = ID | KK of clos_li2li * clos_i2i * int    (* list_int -> list_int clos. d.t. *)
fun g1' (x:int,y:int) : int = + (x,y)                  (* lambda-lifted functions *)
and g2' (x:int) : int = - (x,2)
and id' (p:list_int) : int list = p
and kk' (p:list_int,k:clos_li2li,f:clos_i2i,h:int) : list_int =  (* k,f,h are free vars of kk *)
        let val c1 = dispatch_i2i(f,h)
        in let val c2 = Cons_int(c1,p)
           in dispatch_li2li(k, c2)

and dispatch_i2i(c:clos_i2i,i:int) : int =             (* int->int dispatch function *)
      case c of
        G1 y => g1'(i,y)
      | G2 => g2'(i)
and dispatch_li2li(c:clos_li2li,p:list_int) : list_int =   (* list_int -> list_int disp. fun. *)
      case c of
        ID => id'(p)
      | KK(k,f,h) => kk'(p,k,f,h)

fun outer (y:int,q:list_int) : list_int =
  let val g1:clos_i2i = G1 y                           (* closures *)
  in let val g2:clos_i2i = G2
     in let val id: clos_li2li = ID
        in let fun map(f:clos_i2i,l:int list,k:clos_li2li) : list_int =
                 case l of
                   Nil_int => Nil_int
                 | Cons_int(h,t) =>
                      let val kk:clos_li2li = KK(k,f,h)
                     in map(f,t,kk)
           in map (g1, q, id)
```

Figure 20: Closure conversion at multiple types.

- 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 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.

A detailed specification of the term conversion algorithm is given in Figure 21. The translation of type τ is denoted $\overline{\tau}$. $\mathcal{TS}$ translates top-level declarations, $\mathcal{E}$ 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. $\mathcal{S}$, which is parameterized by a free variable sequence, produces function stubs from functions, as explained below. $\mathcal{A}$ translates mutually recursive sets of algebraic type declarations.

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

i. a mapping *Lift* from source function names to corresponding lifted function names;

ii. an 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 mapping *Stub* from source top-level function names to corresponding stub function names;

v. a set *Lifted* of lifted function definitions; and

vi. a mapping *ClosData* from closure datatype names tc to sets of tuples $(dc, f, \{fv : \tau\})$, where dc is a fresh closure data constructor of tc, f is the corresponding (lifted) function name, and $\{fv : \tau\}$ is the sequence of the corresponding function's free variables and their types.

The mappings *Lift*, *ClosType*, *Dispatch*, and *Stub* are treated as idempotent 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

$$
\begin{aligned}
\overline{K} &= K \\
\overline{D} &= D \\
\overline{(\{\tau\}_*) \to \tau} &= \mathit{ClosType}[\![(\{\overline{\tau}\}_*) \to \overline{\tau}]\!] \\
\\
\mathcal{E}_k[\![se]\!] &= se \\
\mathcal{E}_k[\![v(\{se\}_,)]\!] &= \mathit{if}\ v \in \mathit{Dom}(k) \\
&\ \mathit{then\ let}\ \{fv\}\ \texttt{=}\ k[\![v]\!]\mathit{in}\ \mathit{Lift}[\![v]\!]\ \ (\{se\}_,\texttt{,}\ \{fv\}_,) \\
&\ \mathit{else}\ (\mathit{Dispatch} \circ \mathit{ClosType} \circ \mathit{Typeof})[\![v]\!]\ \ (v\texttt{,}\ \{se\}_,) \\
\mathcal{E}_k[\![c(\{se\}_,)]\!] &= c(\{se\}_,) \\
\mathcal{E}_k[\![p(\{se\}_,)]\!] &= p(\{se\}_,) \\
\mathcal{E}_k[\![\texttt{let val}\ v:\tau\ \texttt{=}\ e_1\ \texttt{in}\ e_2]\!] &= \texttt{let val}\ v:\overline{\tau}\ \texttt{=}\ \mathcal{E}_k[\![e_1]\!]\ \texttt{in}\ \mathcal{E}_k[\![e_2]\!] \\
\mathcal{E}_k[\![\texttt{let fun}\ \{fdec\}_{\texttt{and}}\ \texttt{in e}]\!] &= \mathit{let}\ \{fv\} = \mathcal{FV}_k[\![\texttt{fun}\ \{fdec\}_{\texttt{and}}]\!]\ \mathit{in} \\
&\ \mathit{let}\ k\text{'} = k + (\mathit{FunName}[\![fdec]\!] \mapsto \{fv\})\ \mathit{in} \\
&\ \mathcal{FS}_{k\text{'}}(\{fv\})[\![\{fdec\}]\!][\![e]\!] \\
\mathcal{E}_k[\![\texttt{case}\ se\ \texttt{of}\ \{c(\{v\}_,)\texttt{=>}\ e\}]\!] &= \texttt{case}\ se\ \texttt{of}\ \{c(\{v\}_,)\texttt{=>}\ \mathcal{E}_k[\![e]\!]\} \\
\\
\mathcal{FS}_k(\{fv\})[\![fdec\{fdec\}]\!][\![e]\!] &= \texttt{let}\ \mathcal{F}_k(\{fv\})[\![fdec]\!]\ \texttt{in}\ \mathcal{FS}_k(\{fv\})[\![\{fdec\}]\!][\![e]\!] \\
\mathcal{FS}_k(\{fv\})[\![\ ]\!][\![e]\!] &= \mathcal{E}_k[\![e]\!] \\
\\
\mathcal{F}_k(\{fv\})[\![v\ \ \mathit{inl}\ \ (\{v:\tau\}_,):\tau\ \texttt{=}\ e]\!] &= \mathit{Lifted} := \mathit{Lifted} + \\
&\ \quad (\mathit{Lift}[\![v]\!]\ \mathit{inl}\ (\{v:\overline{\tau}\}_,,\ \{fv:\overline{\mathit{Typeof(fv)}}\}_,):\overline{\tau} = \mathcal{E}_k[\![e]\!]); \\
&\ \mathit{let}\ tc = (\mathit{ClosType} \circ \mathit{Typeof})[\![v]\!]\ \mathit{in} \\
&\ \mathit{let}\ c = \mathit{newDataCon}()\ \mathit{in} \\
&\ \mathit{ClosData} := \mathit{ClosData} + \\
&\ \quad (tc \mapsto (c,\mathit{Lift}[\![v]\!],\{fv:\overline{\mathit{Typeof(fv)}}\})); \\
&\ \texttt{val}\ v:tc\ \texttt{=}\ c(\{fv\}_,) \\
\\
\mathcal{A}[\![\texttt{datatype}\ \{D\ \ \mathit{flt}\ \texttt{=}\ \{c[\texttt{of}\{\tau\}_*]\}_|\}_{\texttt{and}}]\!] &= \{D\ \ \mathit{flt}\ \texttt{=}\ \{c[\texttt{of}\{\overline{\tau}\}_*]\}_|\} \\
\\
\mathcal{TS}_k[\![\texttt{val}\ v:\tau\ \texttt{=}\ e\ \ \mathit{topdecs}]\!] &= \texttt{val}\ v:\overline{\tau}\ \texttt{=}\ \mathcal{E}_k[\![e]\!]\ \ \mathcal{TS}_k[\![\mathit{topdecs}]\!] \\
\mathcal{TS}_k[\![\texttt{fun}\ \{fdec\}_{\texttt{and}}\ \ \mathit{topdecs}]\!] &= \mathit{let}\ \{fv\} = \mathcal{FV}_k[\![\texttt{fun}\ \{fdec\}_{\texttt{and}}]\!]\ \mathit{in} \\
&\ \mathit{let}\ k\text{'} = k + (\mathit{FunName}[\![fdec]\!] \mapsto \{fv\})\ \mathit{in} \\
&\ \{\mathcal{F}_{k\text{'}}(\{fv\})[\![fdec]\!]\}\ \ \{\mathcal{S}(\{fv\})[\![fdec]\!]\}\ \ \mathcal{TS}_k[\![\mathit{topdecs}]\!] \\
\mathcal{TS}_k[\![\ ]\!] &= \\
\\
\mathcal{S}(\{fv\})[\![v\ \ (\{v:\tau\}_,):\tau\ \texttt{=}\ e]\!] &= \texttt{fun}\ \mathit{Stub}[\![v]\!]\ \ (\{v:\tau\}_,)\ \texttt{=}\ v\ \ (\{v:\tau\}_,\texttt{,}\ \ \{fv:\overline{\mathit{Typeof(fv)}}\}_,)
\end{aligned}
$$

Figure 21: Closure conversion of SIL terms.

$$
\begin{aligned}
\mathcal{FV}_k\llbracket \texttt{fun } \{\mathit{fdec}\}_{\texttt{and}} \rrbracket &= \bigcup\{\mathcal{FV}_k\llbracket \mathit{fdec} \rrbracket\} \\
\mathcal{FV}_k\llbracket v \;\; \mathit{inl} \;\; (\{v : \tau\}) : \tau \texttt{ = } e \rrbracket &= \mathcal{FV}_k\llbracket e \rrbracket - \bigcup\{\boldsymbol{\{}v\boldsymbol{\}}\} - \boldsymbol{\{}v\boldsymbol{\}} \\
\mathcal{FV}_k\llbracket (k : K) \rrbracket &= \emptyset \\
\mathcal{FV}_k\llbracket v \rrbracket &= \boldsymbol{\{}v\boldsymbol{\}} \\
\mathcal{FV}_k\llbracket v(\{se\}_{,}) \rrbracket &= \bigcup \mathcal{FV}_k\llbracket se \rrbracket\} \cup (\mathit{if}\ v \in \mathit{Dom}(k)\ \mathit{then}\ k(v)\ \mathit{else}\ \boldsymbol{\{}v\boldsymbol{\}}) \\
\mathcal{FV}_k\llbracket c(\{se\}_{,}) \rrbracket &= \bigcup\{\mathcal{FV}_k\llbracket se \rrbracket\} \\
\mathcal{FV}_k\llbracket p(\{se\}_{,}) \rrbracket &= \bigcup\{\mathcal{FV}_k\llbracket se \rrbracket\} \\
\mathcal{FV}_k\llbracket \texttt{let val } v : \tau \texttt{ = } e_1 \texttt{ in } e_2 \rrbracket &= \mathcal{FV}_k\llbracket e_1 \rrbracket \cup (\mathcal{FV}_k\llbracket e_2 \rrbracket - \boldsymbol{\{}v\boldsymbol{\}}) \\
\mathcal{FV}_k\llbracket \texttt{let fun } \{\mathit{fdec}\}_{\texttt{and}} \texttt{ in } e \rrbracket &= \mathcal{FV}_k\llbracket \texttt{fun } \{\mathit{fdec}\}_{\texttt{and}} \rrbracket \cup (\mathcal{FV}_k\llbracket e \rrbracket - \boldsymbol{\{}\mathit{FunName}\llbracket \mathit{fdec} \rrbracket\boldsymbol{\}}) \\
\mathcal{FV}_k\llbracket \texttt{case } se \texttt{ of } \{c(\{v\})\texttt{=> } e\} \rrbracket &= \mathcal{FV}_k\llbracket se \rrbracket \cup (\bigcup\{\mathcal{FV}_k\llbracket e \rrbracket - \bigcup\{\boldsymbol{\{}v\boldsymbol{\}}\}\})
\end{aligned}
$$

Figure 22: Calculation of free variables. To avoid confusion, we use bold brackets ($\boldsymbol{\{\}}$) to denote sets and ordinary brackets ($\{\}$) to denote syntactic sequences. The notation $\bigcup\{X\}$ denotes the set union of all the sets X resulting from a calculation on members of a syntactic sequence.

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 21 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.

Conversion of top-level functions is complicated by the possibility that they might be exported from the SIL component. The types of exported values must not by changed by any transformation; moreover, exported types should never be closures, since the surrounding 3GL context certainly knows nothing about how to invoke a closure. This implies that the argument and result types of exported functions must not be arrow types (as we already noted in Section 3), and that the exported functions themselves must not be closure-converted. On the other hand, top-level functions might be used as a first-class values within the component, and thus *do* in general need to be closure-converted. The solution is to closure-convert each top-level function, but also introduce a corresponding *stub* function, which has the original function's signature but a new name, and export the stub in place of the original. The body of the stub function simply invokes the closure-converted function using the stub's arguments and the free variables (which are guaranteed to be in the top-level scope at the point of the stub's definition). Stub functions not needed for export will be removed as dead code by the standard optimizer.

The auxiliary function $\mathcal{FV}_k\llbracket e \rrbracket$, specified in Figure 22, computes the free variables of expression e assuming the initial known function environment k. As specified, $\mathcal{FV}$ 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 we actually need the free variables of the *translated* term, but (because of potential recursion) we need them before the translation has been done! To break the circularity, we observe that the free variables sets of source and translated terms can only

```
M[[ export {export} {atdecs} {topdecs}]] =
(  Lift := ∅;  CloseType := ∅;  Dispatch := ∅;  Stub := ∅;
   Lifted := ∅;  ClosData := ∅;
   let {atdec'} =   Flatten {A[[atdecs]]}  in
   let {topdecs'} = TS_∅[[{topdecs}]]  in
   (* at this point all mappings have been built *)
   let {closure_atdec} =
        {tc = {dc of {τ}_*}_|  | tc ∈  Codom(ClosTypes); {(dc, _, {(_, τ)})} ∈  ClosData(tc)}  in
   let {dispatch_fun} = {Dispatch[[tc]] (v_0 : tc, {v : μ}_,) : μ =
                                        case v_0 of
                                          {dc ({fv : τ}) => f({v}_, , {fv}_,)}_|
                           | tc ∈ Codom(ClosTypes); {(dc, f, {fv : τ})} ∈ ClosData(tc);
                            ({μ}_*) -> μ = ClosType^{-1}(tc);   v_0 , {v} fresh }
   let {lifted_fun} = {f |  f ∈ Lifted}  in
   let {export'} = {X[[export]]}  in
   export {export'}
   datatype {atdec'}_and and {closure_atdec}_and
   fun {dispatch_fun}_and and {lifted_fun}_and
   topdecs'
)

X[[ type τ "name"]]  =  type τ "name"
X[[ val v  :  τ "name"]] =
   let v' = if v ∈ Dom(Stub) then Stub[[v']] else v
   in  val v' : τ "name"
```

Figure 23: Closure conversion of SIL components. The notation $\{s \mid s \in S\}$ 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.

differ due to the replacement of a known function application $f(\{v\}_,)$ by the corresponding lifted application $Lift[\![f]\!](\{v\}_,, \{fv\}_,)$, where $\{fv\}$ are the free variables of f. In this case the target free variable set should not include f, but should include the $\{fv\}$.[11]

The top-level conversion function $\mathcal{M}$ for components is shown in Figure 23. 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 `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.

In general, it is possible for freshly-created closure datatype declarations to refer to the converted versions of source program datatype declarations (since free variables may belong to datatypes)

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

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 (i.e., ordinary value declarations, closure value declarations, and stub function declarations) 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.

As a further example, Figures 24 and 25 show the result of applying the conversion algorithm to our running example. For compactness, we omit some datatype and value declarations that are completely unreferenced and hence immediately known to be dead code; the code is otherwise unoptimized. There are two closure datatypes; one (for `point * list_point -> list_point`) describes the possible arguments to `foldl1`; the other (for `list_point -> list_point`) describes the possible results of partially applying `foldl1` (which is curried). Notice that this latter datatype contains closure constructor for `doit` (because it has a matching type) even though `doit` never actually escapes; this constructor will be removed by the standard optimizer. Function `doit''` is the exported stub generated to replace the original `doit`.

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 [28]. Bell, Bellegarde and Hook [5] 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.

We also depend 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. 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

```
exports
  ... val doit'' : list_point -> list_point "doit" ...

datatype pxlp2lp_clos =              (* point * list_point -> list_point closure datatype *)
           Ccons
         | Cconsapp of transform
and lp2lp_clos =                     (* list_point -> list_point closure datatype *)
           Cdoit of list_transform
         | Cg1 of pxlp2lp_clos * list_point

fun dispatch_pxlp2lp (c: pxlp2lp_clos, x:point, l: list_point) : list_point =
    case c of                        (* point * list_point -> list_point dispatch function *)
      Ccons => cons'(x,l)
    | Cconsapp whole_t => consapp'(x,l,whole_t)

and dispatch_lp2lp (c:lp2lp_clos, ps:list_point) : list_point =
     case c of                        (* list_point -> list_point dispatch function *)
       Cdoit(ts) => doit'(ps,ts)
     | Cg1(c',l) => g1'(ps,c',l)

and consapp' (x:point, l:list_point, whole_t: transform) : list_point =  (* free var: whole_t *)
     let val x' : point  = apply (whole_t,x)
     in Cons_point (x',l)

and cons' (x:point, l: list_point) : list_point = Cons_point(x,l)

and f1' (n: list_point, l: list_point, c: pxlp2lp_clos) : list_point =   (* free var: c *)
    case l of
      Nil_point => n
    | Cons_point(x,r) =>
        let n' : list_point = dispatch_pxlp2lp(c,x,n)
        in f1'(n',r,c)

and g1' (l0; list_point, c : pxlp2lp_clos, n: list_point) : list_point = f1'(n,l0,c)
                                                                          (* free vars: c,n *)

and foldl1' (c: pxlp2lp_clos, n: list_point) : pl2pl_clos = Cg1(c,n)

and f0' (n:transform, l:list_transform) : transform =
    case l of
      Nil_transform => n
    | Cons_transform(x,r) =>
       let val n' = compose(x,n)
       in f0' (n',r)
```

Figure 24: Results of closure-converting example program (beginning).

```
and doit' (ps:list_point,ts: list_transform) : list_point =            (* free var: ts *)
     let val whole_t : transform = f0'(id,ts)
     in let val consapp : pxlp2lp_clos = Cconsapp(whole_t)
        in let val f : lp2lp_clos =  foldl1 (consapp,Nil_point)
           in let val ps' : list_point = dispatch_lp2lp(f,ps)
              in let val cons : pxlp2lp_clos = Ccons
                 in let val g : lp2lp_clos = foldl1(cons,Nil_point)
                    in dispatch_lp2lp(g,ps')
...
val ts : list_transform = Cons_transform(...)

fun doit''(ps : list_point) : list_point = doit'(ps,ts)    (* stub function for export *)
```

Figure 25: Results of closure-converting example program (conclusion).

closure-converted program.[12] 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.

Figure 26 shows the effect of optimizing the code in Figure 24. Function `foldl1` has been uncurried, removing the intermediate closures constructions `f` and `g` in `doit'`, and dead code elimination has then removed the `lp2lp_clos` datatype and the associated dispatch function altogether, allowing `g1'` to be eliminated and `doit'` to be inlined into `doit''`. Function `dispatch_pxlp2lp`, having already absorbed `consapp'` and `cons'`, has been inlined into `f1'`. The remaining closure datatype `pxlp2lp_clos` can be represented "flat" and hence need not be heap-allocated. The remainder of this section describes these points in more detail.

10.1 Uncurrying

The general-purpose optimization rules 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:

[12] In C, which supports indirect jumps to top-level functions, we could convert from our representation back to a conventional closure representation as a final compilation step, by choosing the lifted functions' code pointers to *be* the closure type's constructor tags. (This works because each closure value is cased over only once, by the relevant dispatch function.) Of course, we would need to add unsafe casts to the C code.

```
exports
  ...
  val doit'' : list_point -> list_point "doit"
  ...

datatype pxlp2lp_clos flat = Ccons | Cconsapp of transform

fun f1' (n: list_point, l: list_point, c: pxlp2lp_clos) : list_point =
    case l of
      Nil_point => n
    | Cons_point(x,r) =>
         case c of
           Ccons =>
             let val n' :list_point = Cons(x,n)
             in f1'(n',r,c)
         | Cconsapp(whole_t) =>
             let val x' : point = apply(whole_t,x)
             in let val n' : list_point = Cons(x',n)
                in f1'(n',r,c)

fun f0' (n:transform, l:list_transform) : transform =
    case l of
      Nil_transform => n
    | Cons_transform(x,r) =>
        let val n' = compose(x,n)
        in f0' (n',r)

...
val ts : list_transform = Cons_transform(...)

fun doit''(ps : list_point) : list_point =
  let val whole_t = f0'(id,ts)
  in let val consapp : pxlp2lp_clos = Cconsapp(whole_t)
     in let val ps' : list_point = f1'(Nil_point,ps,consapp)
        in f1'(Nil_point,ps', Ccons)
```

Figure 26: Optimized first-order code.

```
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 [1] 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:

```
datatype t2_t_clos = Cf2 of t1 | ...

fun f'(x1:t1) : t2_t_clos = Cf2(x1)

fun t2_t_dispatch (c:t2_t_clos, x2:t2) : t =
   case c of
     Cf2 x1 => f2'(x2,x1)
   | ...

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

let val g1 : t2_t_clos = f'(e1)
in t2_t_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 : t2_t_clos = Cf2(e1)
in t2_t_dispatch(g1,e2)
```

Now the call to `t2_t_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 `fn`-expressions that might be invoked at each application site in the program. Existing implementations of this so-called *closure analysis* use an abstract interpretation involving a fixpoint calculation [36, 37] or a constraint-based mechanism [9, 30]. Surprisingly, closure analysis does not appear to have been implemented for typed languages, despite the fact that typing obviously provides a good first cut at the analysis "for free." Also, existing closure analysis algorithms do not express their results within 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; only lambdas of that type can possible be invoked at that site, and set of such lambdas is 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 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 [18]. 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 [10, 1] 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.

The payoff from these optimizations depends on the precision of the underlying type-based closure analysis, and this in turn depends on source program types. To the extent that these types represent structural distinctions among values, they are essentially fixed by the 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 farenheit = F of integer
datatype centigrade = C of integer
```

Ordinarily, programmers do this in order to make their program text clearer and to obtain help from the compiler's typechecker in detecting logical errors. For example, lambda-bound functions of type `farenheit -> farenheit` can be reliably distinguished from those of type `centigrade -> centigrade`, etc., reducing the risk of accidentally confusing the two kinds of quantities. 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, and produce 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 we already have the typed intermediate form in hand, and 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 [9] and further analyzed by Palsberg [30].

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 $\mathtt{D}^i$ `=` $\mathtt{C}^i$ `of` τ_2 and replaces all instances of $\tau_1 {\rightarrow}^i \tau_2$ by $\tau_1 \rightarrow D^i$, adding the necessary coercions to the program. These amount to a $\mathtt{C}^i$ construction around the body of each function of this type and a `case` on the result of each application of such a function. The resulting program is fed directly to the ordinary closure converter.

The coercions just mentioned carry no runtime cost (since the $\mathtt{D}^i$ are "transparent" constructors), but they do inhibit some further SIL optimizations. We have therefore developed a clean-up transformation, to be applied *after* closure conversion, that gets rid of all transparent datatypes.[13]

As ongoing work, we are trying 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.

[13] In fact, it may prove useful to get rid of all non-structural typing distinctions at this point, rewriting the program to use a set of structurally-distinct canonical datatypes. We have yet to implement this idea.

11 Eliminating Tail Calls

Function calls are generally expensive in standard implementations of our target 3GLs.[14] So it is valuable to avoid making recursive calls where possible, and it is particularly 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 [20]. Jump points are declared similarly to local functions, with a label name, formal parameters, defining expression, and scoped-over body expression; `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
- the scope of a jump point label does *not* extend into function declarations nested inside the body expression.

These restrictions guarantee that the target label of a `goto` is always in the same function as the `goto` itself. Hence, when SIL is translated to 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 to a set of assignments to the parameter variables followed by an ordinary 3GL local `goto`. Here's an example in SIL together with the corresponding C code:

```
fun f(x:int) : int =
  label g(y:int) = y+1
  in let val b : bool = x > 8
     in case b of
          true => g x
        | false => g 0
```

```
int f(int x)
{ int y;
  {int b = x > 8;
   switch (b) {
     case 1: y = x;
             goto g;
     case 0: y = 0;
             goto g;
   }
  };
g: return (y+1);
}
```

Because `goto`s must be in tail position, code that uses jump points is not amenable to simplifying rewrites like inlining. Therefore, we introduce such code only at the very last minute, after first-order conversion and all optimizations are completed, and just before conversion to MIL.

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

```
fun f0' (n:transform, l: list_transform) : transform =
  let label jp0 (n0:transform, l0:list_transform) =
    case l0 of
      Nil_transform => n0
    | Cons_transform(x,r) =>
        let val n' = compose(x,n0)
        in goto jp0 (n',r)
  in goto jp0 (n,l)

...

fun doit''(ps : list_point) : list_point =
  let val whole_t = f0'(id,ts)
  in ...
```

Figure 27: Insertion of jump points into example code.

When can a tail-call be turned into a `goto`? Since all functions have been lifted to top level at this point, at least one call to every (non-dead) function must be from the body of *different* top-level declaration. Thus, it is never possible to convert a function definition and its calls directly into a jump point and corresponding `goto`s; the original function must be preserved for the sake of the external (non-recursive) caller. But we are free to introduce a jump point at the top of the function body, for use by recursive tail calls; external and non-tail recursive calls continue to use the original function. As an example, Figure 27 shows how a jump point is introduced at the top of `f0'` in the code of Figure 26.

This approach works well for removing simple tail-recursion; the only added cost is the extra `goto` associated with calls to the original function.[15] 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

[15] This expense could be reduced by a simple algorithm for ordering code blocks; decent 3GL compilers already do this.

```
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 Imperative Code

12.1 MIL

The translation of first-order, optimized SIL code into our target 3GLs is mediated by a translation to a common imperative intermediate form called MIL, whose syntax is given in Figure 28. MIL is built around imperative *statements* in which assignments update the values of variables. In particular, the body of each SIL function becomes a MIL statement, which in turn becomes the body of the corresponding 3GL function. Statements are defined recursively in such a way that they group into *sequences*. In addition to assignments, sequences may contain `case` statements that dispatch on the tag of an algebraic type, or nested statement `block`s. Each sequence terminates with an explicit `return`, which exits from the enclosing function, or with a `goto` to some locally defined label; sequences never "fall through" one to another. A `block` is used to declare a set of local variables and (mutually recursive) labeled statement sequences. The block

`block var` $v_1 : \tau_i \ldots\ v_n : \tau_n$ `lab` $l_1 : st_1 \ldots\ l_m : st_m$ `begin` st

corresponds to these C and Ada blocks

(types)	τ ::=	K	(primitive types)
	|	D	(monomorphic algebraic types)
(simple expressions)	se ::=	`(`$k : K$`)`	(primitive constants)
	|	v	(variables)
	|	$v.a$	(record selections)
(expressions)	e ::=	se	(simple expressions)
	|	$f(\{se\}_,)$	(function applications)
	|	$c(\{se\}_,)$	(constructor applications)
	|	$p(\{se\}_,)$	(primitive applications)
(statements)	st ::=	`return` se	
	|	v `:=` e `;` st	
	|	`goto` l	
	|	`case` v `of` $\{c$ `=>` $st\}_{\|}$	
	|	`block var` $\{vdec\}$ `lab` $\{jdec\}$ `begin` st	
(variable declarations)	$vdec$ ::=	$v : \tau$	
(function bindings)	$fdec$ ::=	$f(\{v : \tau\}) : \tau = st$	
(labeled statement bindings)	$jdec$ ::=	$l : st$	
(mutually recursive decls.)	$atdecs$ ::=	`datatype` $\{atdec\}_{\texttt{and}}$	
(algebraic type decls.)	$atdec$ ::=	$D = rep$ $\{c$ `of` $(\{a : \tau\})\}$	
(algebraic type representations)	rep ::=	`enum`	
	|	`value`	
	|	[`flat`] `record`	
	|	[`flat`] `variantrecord`	
	|	`oneNull` c rep	
(exports)	$export$ ::=	`type` τ "*name*"	
	|	`val` $v : \tau$ "*name*"	
(components)	m ::=	`export` $\{export\}$ $\{atdecs\}$ $\{vdecs\}$ $\{fdec\}$ st	

Figure 28: MIL Syntax. Note that all *jdec*s in a given let are treated as mutually recursive, as are all *fdec*s in a component.

```
{τ_1 v_1; ...; τ_n v_n;
 st;
 l_1:
   st_1
 ...
 l_m:
   st_m
}
```

```
DECLARE
 v_1 : τ_1; ...; v_n : τ_n;
BEGIN
 st;
 <<l_1>>
   st_1
 ...
 <<l_m>>
   st_m
END;
```

Variables do *not* have associated initializing expressions; legal programs must take care to initialize any variable before using it.

Argument and return values must be *simple expressions*, i.e., constants, simple variables, or dereferenced fields of constructed values (see below); application expressions can appear only as the right-hand sides of assignments. In particular, this guarantees that primitive applications appear only in the context of assignments v `:=` $p(\{se_,\})$, so there will always be a suitable target variable into which the primitive's template code can store the result.

Algebraic data types are specified in more detail than in RML or SIL; their declarations include representation information (see Section 12.2) and the individual fields of each constructor are named and can be dereferenced using dot notation. Such dereferences can legally occur only within an appropriate arm of `case` over some value of the corresponding data type.

The top level of a MIL component consists of a list of exports; a set of algebraic type declarations, which must include definitions of all exported types; a set of variable declarations and a (mutually recursive) set of function declarations, which between them must include definitions of all exported values; and an initializing statement, which fills in the values of the top-level variables and which the surrounding 3GL driver must arrange to execute (once) before any exported value is used.

A slightly simplified formulation of the translation of SIL to MIL is specified in Figure 29. The core of the translation scheme is the function $\mathcal{E}(v_0, st_0)[\![e]\!]$, which converts the SIL expression e into a MIL statement, in a context where the expression should be assigned to variable v_0, and the immediately *following* statement should be st_0. Thus, statement sequences are built up in reverse order. The root argument for st_0 is either a `return` statement, as specified in the $\mathcal{FS}$ rule, which translates functions, or a `goto` statement.

Each SIL *ldec* generates a MIL block with local variables corresponding to the *ldec*'s arguments and a labeled statement sequence corresponding to the *ldec*'s body. The translation of a corresponding SIL `goto` (which must always be in tail position) is a parallel assignment of the actual parameter expressions to the variables representing the formal parameters, terminated by a MIL `goto` to the label; the st_0 parameter is ignored. Since the actual parameters may refer to the current values of the formal parameters, it is in general necessary to introduce new temporary names for the actual values; once the temporaries have all been defined, they can be safely used to overwrite the formals. Here and elsewhere, the translation as presented is rather profligate in its generation of fresh variables; in practice we use a somewhat more complex translation that avoids

$$
\begin{array}{rcl}
\mathcal{E}(v_0, st_0)[\![se]\!] & = & v_0 \ \texttt{:=}\ se\texttt{;}\ st_0 \\
\mathcal{E}(v_0, st_0)[\![f(\{se\}_,)]\!] & = & v_0 \ \texttt{:=}\ f(\{se\})\texttt{;}\ st_0 \\
\mathcal{E}(v_0, st_0)[\![c(\{se\}_,)]\!] & = & v_0 \ \texttt{:=}\ c(\{se\})\texttt{;}\ st_0 \\
\mathcal{E}(v_0, st_0)[\![p(\{se\}_,)]\!] & = & v_0 \ \texttt{:=}\ p(\{se\})\texttt{;}\ st_0 \\
\mathcal{E}(v_0, st_0)[\![\texttt{let}\ v:\tau\ \texttt{=}\ e_1\ \texttt{in}\ e_1]\!] & = & \texttt{block var}\ v:\tau\ \texttt{begin}\ \mathcal{E}(v, \mathcal{E}(v_0, st_0)[\![e_2]\!])[\![e_1]\!] \\
\mathcal{E}(v_0, st_0)[\![\texttt{case}\ v\ \texttt{of}\ \{calts\}]\!] & = & \\
 & & \texttt{block lab}\ l: st_0\ \texttt{begin case}\ v\ \texttt{of}\ \{\mathcal{R}(v, v_0, \texttt{goto}\ l)[\![calt]\!]\} \\
 & & \hfill (l \text{ fresh}) \\
\mathcal{E}(v_0, st_0)[\![\texttt{let label}\ ldecs\ \texttt{in e}]\!] & = & \\
 & & \texttt{block lab}\ l: st_0\ \texttt{begin block}\ \mathcal{LS}(v_0, \texttt{goto}\ l)[\![ldecs]\!]\ \texttt{begin}\ \mathcal{E}(v_0, \texttt{goto}\ l)[\![e]\!] \\
 & & \hfill (l \text{ fresh}) \\
\mathcal{E}(v_0, st_0)[\![\texttt{goto}\ l(\{se\}_,)]\!] & = & \texttt{block var}\ \{v:\tau\}\ \texttt{begin}\ \{v\ \texttt{:=}\ se\texttt{;}\}\ \{x\ \texttt{:=}\ v\texttt{;}\}\ \texttt{goto}\ l \\
 & & \hfill (\{v\} \text{ fresh; } \{x:\tau\} = Args(l)) \\
 & & \\
\mathcal{R}(v_1, v_0, st_0)[\![c(\{v\}_,)\ \texttt{=>}\ e]\!] & = & c\ \texttt{=> block var}\ \{v:\tau\}\ \texttt{begin}\ \{v\texttt{:=}v_1.a\texttt{;}\}\ \mathcal{E}(v_0, st_0)[\![e]\!] \\
 & & \hfill (\{a:\tau\} = Fields(c)) \\
 & & \\
\mathcal{LS}(v_0, st_0)[\![\{l(\{v:\tau\}_,):\tau = e\}_{\texttt{and}}]\!] & = & \texttt{var}\ \{\{v:\tau\}\}\ \texttt{lab}\ \{l: \mathcal{E}(v_0, st_0)[\![e]\!]\} \\
 & & \\
\mathcal{FS}[\![\texttt{fun}\ \{f\ \ inl\ (\{v:\tau\}_,):\tau\ \texttt{=}\ e\}_{\texttt{and}}]\!] & = & \\
 & & \{f(\{v:\tau\}_,):\tau\ \texttt{= block var}\ v_0:\tau\ \texttt{begin}\ \mathcal{E}(v_0, \texttt{return}\ v_0)[\![e]\!]\} \\
 & & \hfill (v_0 \text{ fresh}) \\
 & & \\
\mathcal{TS}[\![\texttt{val}\ v:\tau\ \texttt{=}\ e\ \ topdecs]\!] & = & let\ (vs, fs, st)\ \texttt{=}\ \mathcal{TS}[\![topdecs]\!]\ \ in\ (v:\tau\ \ vs\texttt{,}\ fs\texttt{,}\ \mathcal{E}(v, st)[\![e]\!]) \\
\mathcal{TS}[\![fdecs\ \ topdecs]\!] & = & let\ (vs, fs, st)\ \texttt{=}\ \mathcal{TS}[\![topdecs]\!]\ \ in\ (vs\texttt{,}\ \mathcal{FS}[\![fdecs]\!]\ fs\texttt{,}\ st) \\
\mathcal{TS}[\![\ ]\!] & = & (\ ,\ ,\texttt{return null}) \qquad (\texttt{null} \text{ an arbitrary constant})
\end{array}
$$

Figure 29: Transformation of SIL expressions and declarations to MIL statements. Auxiliary function *Args* returns and names and types of the formal parameters to a label, and *Fields* returns the names and types of the fields of a data constructor.

making unnecessary copies of variables, but in any case we assume that the 3GL compiler will do a good job of coalescing unnecessary copies.

The translations for SIL `case` and `let` *ldecs* expressions must arrange to perform the "next statement" st_0 in each subexpression. This aim could be achieved simply by duplicating st_0 everywhere it is needed, but at the potential cost of a code explosion if st_0 is large. Therefore, the translation generates a fresh "join point" label for each such expression, and arranges for each subexpression to jump to it. In practice, we use a slightly more complex translation that avoids generating such join points when st_0 is simple enough that it can be duplicated without danger of a code explosion, which is often the case; this optimization avoids generating lots of unnecessary jumps to jumps.

The $\mathcal{DS}$ rules describe how to translate the top-level declarations of a SIL component into the corresponding MIL sequence of variable declarations, function declarations, and initializing statement. The primary complication arises from the need to separate out the declaration and initializing definition of each SIL value declaration.

Figure 30 shows the MIL equivalent to the function `f0'` from Figure 27, just as it would be generated by the translation rules described here (without optimizations). The equivalent C code is shown in Figure 31. Figure 6 shows similar Ada code, based on MIL code generated with the optimizations mentioned above.

12.2 Choosing Representations for Algebraic Types

Any algebraic type can be given a default representation 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. Such types can be defined in a straightforward manner in each target 3GL. However, many types can be given much more efficient representations [1, 10]. In particular, it is often possible to avoid boxing small records, since in a monomorphic setting there is no reason to require that all types be representable in a single word. The only significant restriction on our choice of representations is that they must be typable in the target 3GL.

Our translator automatically chooses optimized target-language representations in the following cases, which correspond to the possible values of *rep* in the MIL syntax description.

- If no data constructor carries values, an (unboxed) enumerated type is used.
- If there is only one data constructor, a simple (untagged) boxed record can be used. Moreover, if the data constructor carries only a single value, there is no need to box a singleton record; the value itself can be used.
- If there is exactly one *nullary* data constructor (one that doesn't carry a value), and the other constructors require a boxed representation, the nullary constructor can be represented by the null pointer.[16] *Any* nullary constructor can be represented by a pointer to a statically-allocated address.
- If a type that would normally be boxed is non-recursive and its values occupy a sufficiently small space, it can be represented as unboxed (or *flat*), i.e., manipulated directly by value rather than being heap-allocated and manipulated by reference. The user can also explicitly mark datatypes in the source program as `flat`.

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,

[16] When generating C, we might be tempted to extend this trick [1] to represent multiple nullary constructors as distinct "small" integers (i.e., integers that cannot be confused with pointers), but this would require casting.

```
fun f0 (n: transform, l: list_transform) : transform =
  block
  var n9 : transform
  lab JoinPoint0:
        return n9
  begin
     block
     var n0: transform l0: list_transform
     lab JumpPoint0:
           block
           lab JoinPoint1:
                 goto JoinPoint0
           begin
             case l0 of
               Cons_transform =>
                 block
                 var x: transform r: list_transform
                 begin
                   x := l0.Cons_list_0; r := l0.Cons_list_1;
                      block
                     var n5 : transform
                     begin
                       n5 := compose(x,n0);
                       block
                       var n4: transform l4 : list_transform
                       begin
                        n4 := n5; l4 := r;
                        n0 := n4; l0 := l4;
                        goto JumpPoint0
             | Nil_transform =>
                 block
                 begin
                   n9 := n0;
                   goto JoinPoint1
     begin
       block
       var n7 : transform l7 : list_transform
       begin
         n7 := n; l7 := l;
         n0 := n7; l0 := l7;
         goto JumpPoint0
```

Figure 30: MIL code for example function.

```
transform f0 (transform n, TransList l) {
  transform n9;
  { transform n0; TransList l0;
    { transform n7; TransList l7;
      n7 = n; l7 = l;
      n0 = n7; l0 = l7;
      goto JumpPoint0;
    };
  JumpPoint0:
    { if (l0 != NULL)
        { transform x; TransList r;
          x = l0->Cons_list_0; r = l0->Cons_list_1;
          { transform n5;
            n5 = compose(x,n0);
            { transform n4; TransList l4;
              n4 = n5; l4 = r;
              n0 = n4; l0 = l4;
              goto JumpPoint0;
            }
          }
        }
      else
        { n9 = n0;
          goto JoinPoint1;
        };
    };
    JoinPoint1:
      goto JoinPoint0;
  };
JoinPoint0:
  return n9;
}
```

Figure 31: C equivalent of MIL code for example.

as each move requires that the entire contents of the record be copied. Thus use of the unboxed representation obviously should be restricted to fairly small records; we make the threshold size a tunable parameter of the translator.

Our translator supports unboxed representations even for variant records, unlike other functional language compilers known to us. 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.

One potential advantage of using unboxed values is that they need not, in principle, be 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 14.

13 Generating Ada or C

Since MIL represents a "lowest common denominator" of ANSI C and Ada83, translation to these languages is quite straightforward. If C is the target, the MIL component is translated into a single file containing one top-level declaration for each component value and function, and a special function `Initialize` containing the component-level statement code. If Ada is the target, the MIL component generates an Ada package in two files: a package specification file, which contains the definitions of exported types, the definitions of exported values, and the declaration signatures of exported functions; and a package body file, which contains the definitions of all functions and the initialization code.

We have used the `gcc` compiler for ANSI C compilation and the Sun/Verdix Ada compiler (version ??) 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 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 epect.

We have also had to deal with a number of complications arising from arbitrary limitations in the Verdix 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.

14 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 [34] 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 arrays of reals. `interpd`

		life	fft	interpd	interpc
	line count	302	237	113	119
smlnj	time (sec)	2.0	5.8	2.3	3.0
standard	time (sec)	0.8	2.7	2.0	4.7
	max closure size (words)	3	1	1	1
flow-flat	time (sec)	0.8	2.7	1.7	4.6
flow-boxed	time (sec)	1.0	2.7	2.0	5.1
	max closure size (words)	3	1	5	7
flow-flat-nogc	time (sec)	0.5	2.6	1.2	2.4
	heap alloc (MB)	2.4	0	6.6	29.9
flow-boxed-nogc	time (sec)	1.6	2.7	1.3	2.6
	heap alloc (MB)	3.9	0	8.5	34.9

Table 1: Benchmark results.

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 [8].

All tests were performed on a 133MHz Pentium processor with 80MB memory running under Linux. Row `smlnj` represents the behavior of Standard ML of New Jersey version 109.27, with the compilation settings `reducemore := 0` and `rounds := 0` to encourage thorough optimization of small programs. The other rows represent the behavior of our compiler generating C under a variety of compilation settings; the resulting C was then compiled using `gcc` version 2.7.2.1 with option `-O3`, and (unless otherwise noted) linked with the Boehm-Demers conservative garbage collector [7] version 4.11. 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.

`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 sometimes worthwhile (e.g., for `interpd`), but only in conjunction with the flat representation for closure types.[17]

These comparisons of flat vs. boxed closure representations may be skewed by our use of the relatively slow Boehm-Demers 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`.

[17]The apparent anomaly of `interpc`'s *increased* execution time under `flow-boxed` is due to the optimizer's failure to clean up fully after the introduction of the artificial datatypes that encode the closure analysis rsult.

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.

15 Conclusions

Versions of the compiler described here have been in use within our overall translation system for nearly 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 our approach is a promising alternative to existing 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 have tried to construct our compiler from the best known technologies for FL compilation. We have found the staged approach to compilation very effective for managing a complex group of transformations. Like other researchers [41, 32] we have found the ability to type-check intermediate representations invaluable in uncovering bugs in the course of compiler development. Moreover, we have developed new uses for type information in late-stage optimization of programs.

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

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

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

[3] A. W. Appel and T. Jim. Shrinking lambda expressions in linear time. *Journal of Functional Programming*, 1997. (to appear).

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

[5] J. M. Bell, F. Bellegarde, and J. Hook. Type-driven defunctionalization. In *Proc. 2nd International Conference on Functional Programming*, June 1997.

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

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

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

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

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

[11] L. Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8:147–172, 1987.

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

[13] W.-N. Chin and J. Darlington. Higher-order removal: A modular approach. Unpublished work, 1993.

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

[15] O. Danvy and A. Filinski. Representing control, a study of the cps transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

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

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

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

[19] M. P. Jones. Partial evaluation for dictionary-free overloading. Technical Report YALEU/DCS/RR-959, Yale University Dept. of Computer Scinece, Apr. 1993.

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

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

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

[23] J. L. Lawall and O. Danvy. 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, Jan. 1993.

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

[25] X. Leroy. *The Caml Light System.* INRIA, 0.7 edition, 1995.

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

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

[28] G. Morrisett. *Compiling with Types.* PhD thesis, Carnegie Mellon University, Dec. 1995. Available as TR CMU-CS-95-226.

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

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

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

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

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

[34] C. Reade. *Elements of Functional Programming.* Addison-=Wesley, 1989.

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

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

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

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

[39] D. Tarditi. Design and implementation of code optimizations for a type-directed compiler for Standard ML, Dec. 1996. Technical Report CMU-CS-97-108.

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

[41] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. 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, June 1996.

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

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

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

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

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