JFP **13** (3): 677–708, May 2003. © 2003 Cambridge University Press
DOI: 10.1017/S095679680200463X Printed in the United Kingdom

Compiling for template-based run-time code generation

FREDERICK SMITH, DAN GROSSMAN, GREG MORRISETT
Dept. of Computer Science, 4130 Upson Hall, Cornell University, Ithaca, NY 14853
(*e-mail:* {*fms,danieljg,jgm*}*@cs.cornell.edu*)

LUKE HORNOF
Transmeta Corporation, 3940 Freedom Circle, Santa Clara, CA 95054
(*e-mail: hornof@transmeta.com*)

TREVOR JIM
AT&T Research, 180 Park Avenue, PO Box 971, Florham Park, NJ 07932
(*e-mail: trevor@research.att.com*)

Abstract

Cyclone is a type-safe programming language that provides explicit run-time code generation. The Cyclone compiler uses a template-based strategy for run-time code generation in which pre-compiled code fragments are stitched together at run time. This strategy keeps the cost of code generation low, but it requires that optimizations, such as register allocation and code motion, are applied to templates at compile time. This paper describes a principled approach to implementing such optimizations. In particular, we generalize standard flow-graph intermediate representations to support templates, define a mapping from (a subset of) Cyclone to this representation, and describe a dataflow-analysis framework that supports standard optimizations across template boundaries.

1 Introduction

Cyclone[1] (Hornof & Jim, 1999) is a type-safe language that supports explicit run-time code generation (RTCG). What distinguishes the Cyclone implementation from other RTCG systems, such as Tempo (Noël *et al.*, 1996; Consel & Noël, 1996), DyC (Auslander *et al.*, 1996; Grant *et al.*, 1999; Grant *et al.*, 1997), and ‘C (Engler *et al.*, 1996; Poletto *et al.*, 1997), is that it is a *certifying compiler*: the compiler produces typed assembly language (Morrisett *et al.*, 1999) as output. This output can be mechanically verified for certain safety and consistency properties. In particular, we can statically verify that any dynamically generated code will be type-correct. Such strong guarantees are particularly useful in the RTCG setting where debugging programs and compilers is notoriously difficult.

The original Cyclone compiler used a template-based compilation strategy for

[1] The version of Cyclone discussed in this paper is the precursor to a new Cyclone language being developed at Cornell. That language does not yet support RTCG.

RTCG similar to that used by Tempo and DyC. A *template* is a pre-compiled code fragment. At run time, templates are copied to create a new function. The generating code controls the order and number of the copies, and it overwrites certain operands in the template, called *holes*, with new values. For example, a jump target is often overwritten with an address in the newly generated function.

With this template-based approach, we were able to improve the performance of a number of small benchmarks. However, unlike DyC and Tempo, our original compiler did not perform optimizations such as register allocation, loop-invariant removal, copy propagation, or common-subexpression elimination. Rather, it generated naive, stack-based typed assembly language, and as a result, the absolute performance relative to an optimizing compiler was disappointing.

It became clear that to achieve better performance we needed to optimize the template code more aggressively. In particular, we needed to perform optimizations such as register allocation *across* template boundaries at compile time, even without knowing exactly how templates would be stitched together at run time. Modifying our optimizing compiler for this purpose seemed daunting. Other compilers relied on ad hoc techniques (in the case of Tempo) or extremely sophisticated analyses (in the case of DyC) that were difficult to adapt.

The goal of this paper, therefore, is to describe the framework we developed and the subtleties we encountered in combining traditional optimizations with explicit (programmer-provided) RTCG. To this end, we describe a simple compiler using template-based RTCG and show how we apply traditional dataflow analyses (such as liveness analysis) and optimizations (such as register allocation) to templates. Although our compiler is certifying the framework we develop is not specific to certifying compilers. For this reason we present our work in an untyped context.

The primary contribution of our framework is its simplicity. The understanding developed here should make it easier to add template-based RTCG to an existing optimizing compiler. Thus, our contributions include:

- the design of a block-based intermediate language (CIR) that supports optimization in the presence of RTCG
- a precise translation from a source language with explicit RTCG into CIR
- a discussion of how to adapt existing analyses and transformations to CIR

We have implemented a translation from the full Cyclone language to a CIR-based intermediate form and a few important optimizations (notably inlining, null-check elimination, and graph-coloring intraprocedural register allocation). We found our framework invaluable for understanding and structuring the code. The compiler is available at `http://www.cs.cornell.edu/talc`.

Before proceeding to our focus, we review previous approaches to RTCG (section 2) and present some preliminary performance results for our compiler (section 3). In section 4, we give an informal overview to our compilation and optimization approach. The next four sections make this discussion more precise by presenting a small source language (section 5) and target language (section 6), a complete translation from the source to the target (section 7), and dataflow analyses and optimizations over the target (section 8). Finally, section 9 discusses future work.

https://doi.org/10.1017/S095679680200463X Published online by Cambridge University Press

2 Overview of RTCG and related work

Programming languages like Lisp and Scheme have long had a simple model of RTCG. In this compilation model, the compiler is included in every executable. At run time, programs use `eval` to pass arbitrary "strings" (S-expressions actually) to the compiler for compilation and the resulting function is dynamically linked so that it is immediately available. For greater flexibility, these languages also support a comma operator, which embeds run-time values into the generated program.

By using the compiler to dynamically generate code, this approach ensures that the dynamic code can be well optimized. For certain kinds of applications, full optimization may be the only way to achieve significant improvements and the time spent compiling may be easily recovered. However, for many simple functions a naive compiler can generate optimal code and this time will be wasted. In other cases, a generated function may be too little used to recoup the time spent compiling.

In contrast to this heavyweight approach, lightweight RTCG aims to minimize dynamic compile times while retaining code quality. One approach to minimizing compile times is to use fast but less precise linear-optimization techniques. The 'C compiler takes this approach. 'C (Engler *et al.*, 1996; Poletto *et al.*, 1997) is an unsafe language providing rich low-level control over RTCG. The compiler for 'C, `tcc`, is a modified version of `lcc` (Fraser & Hanson, 1995) that lets users choose among different run-time compilers to trade code quality for compile time.

The cost of dynamic compilation can also be reduced by performing optimizations in advance at compile time. To make this approach practical, it is useful to limit the set of functions that can be generated at each code-generating program point. By restricting the set of possible functions, the compiler gains enough context to optimize the code effectively although the programmer may lose some flexibility and convenience.

The template-based strategy to RTCG, pioneered by Tempo and the precursor to DyC, is an extreme case of lightweight RTCG. In this approach, which the Cyclone compiler uses, the dynamic compiler simply copies pre-compiled code fragments together and replaces jump targets and instruction operands as appropriate. Previous work has shown that this approach can generate code hundreds to thousands of times faster than a conventional compiler while retaining high code quality for a range of applications (Noël *et al.*, 1996; Auslander *et al.*, 1996).

Tempo (Noël *et al.*, 1996; Consel & Noël, 1996) is a partial-evaluation system for C that can be configured for RTCG. Based on user annotations, Tempo applies a binding-time analysis to stage the program into dynamic and static parts. Instead of generating machine code directly like the Cyclone compiler, Tempo uses an off-the-shelf C compiler (`gcc` or `lcc`) as a back end. All code, including template code, is compiled by constructing C source code that is then passed through the C compiler. Tempo gets inter-template optimizations by cleverly arranging a set of related source code templates into the body of a single dummy function and extracting the machine code templates from the body of the compiled function. In essence, Tempo implements inter-template optimizations using the back-end compiler's intra-function optimizations. The disadvantage of this approach is that the intra-function

https://doi.org/10.1017/S095679680200463X Published online by Cambridge University Press

optimizations may result in unsound inter-template optimizations (Noël *et al.*, 1996). Knowing when this will happen requires intimate knowledge of how Tempo compiles templates and how the back-end compiler performs intra-function optimizations. The system performs correctly most of the time, but when it breaks, it takes an expert to debug it. Our framework supports sound inter-template optimizations directly; in principle Cyclone could serve as a more robust back end for the Tempo system.

DyC (Auslander *et al.*, 1996; Grant *et al.*, 1999; Grant *et al.*, 1997) is a modified version of the Multiflow (Lowney *et al.*, 1993) optimizing C compiler that accepts user annotations to direct a binding-time analysis. DyC uses templates internally but as a final pass emits code that performs simple local optimizations such as peep-hole optimizations and strength reduction at dynamic compile time. Another significant difference between DyC and Cyclone is that DyC delays explicitly staging the user code until *after* most traditional optimizations. By keeping the staging implicit in the annotations on variables, the compiler gains some simplicity but the user loses control over the code that is dynamically generated. Because we have found that the performance advantage of RTCG is often brittle, Cyclone has programmers explicitly stage their programs and the compiler optimizations never change the explicit staging.

There are a number of other RTCG projects. Fabius (Leone & Lee, 1994; Lee & Leone, 1996) pioneered using generic generating extensions for RTCG (they have long been used in partial evaluation). A generating extension is a function taking in a compilation context and producing code. PML (Wickline *et al.*, 1998; Davies & Pfenning, 1996) grew out of the theoretical work that showed the connection between modal logics and RTCG. PML compiles into the Categorical Abstract Machine (Cousineau *et al.*, 1985) augmented with RTCG instructions. MetaML (Taha & Sheard, 1997; Taha, 2000) is an extension of ML based on modal type systems that allows safe, explicit RTCG. The MetaML compiler relies on a heavyweight approach to RTCG.

3 Preliminary experimental results

Although this paper is about a framework for compilation and not about performance, we present some preliminary results for our compiler. Previous work (Noël *et al.*, 1996; Consel & Noël, 1996; Grant *et al.*, 1999; Auslander *et al.*, 1996) has examined the merits of template-based RTCG. Although we think there is more to say on this subject, it is not our goal to do so here. Instead, we hope to illustrate the benefits of RTCG and the effectiveness of our compiler.

We present results for eight numerical micro-benchmarks that have been studied previously, and for a port of the functional simulator (`sim-fast`) from the Simple Scalar Toolset (version 1.0) (Burger *et al.*, 1996). All experiments were performed on a 750-Mhz Pentium II with 256 MB of memory running Linux RedHat 7.0. We measured running times using the Pentium's cycle counter to obtain cycle-level precision for the micro-benchmarks and `gettimeofday` for the simulator runs. The micro-benchmarks were run ten times. The simulator was run three times on each input. We present the best running time in each case.

https://doi.org/10.1017/S095679680200463X Published online by Cambridge University Press

Table 1. *A brief description of the micro-benchmarks. The Specialized column indicates the value used to dynamically generate code. The LOC code shows the lines of code for the key function*

Benchmark	Description	Specialized	LOC
cheb	Chebyshev approximation	Degree of the polynomial	13
csi	Cubic spline computation	X-coordinates	32
dot	Integer dot product	One input vector, 20% zeroes	8
dotf	Floating-point dot product	One input vector, 20% zeroes	8
fft	Fast-fourier transform	Size of the input	59
poly	Compute an integer polynomial	Coefficients	12
polyf	Compute a floating-point polynomial	Coefficients	12
romberg	Romberg integration	Interval and precision	34

3.1 Micro-benchmarks

Table 1 gives a brief description of the micro-benchmarks and how they were specialized. Each micro-benchmark is less than 60 lines of code. Of these programs, `cheb`, `csi`, `dot`, `fft` and `romberg` have previously been used for performance measurements by Tempo (Noël *et al.*, 1996).

For each micro-benchmark, we wrote four versions: (No RTCG) a version without RTCG, (RTCG) a version with RTCG, (unsafe-C) a naive C version, and (safe-C) a C version using the same data representations and explicitly performing all the bounds-checks that the Cyclone version performs. (Recall Cyclone is a type-safe language; our compiler does not yet implement bounds-check elimination.) The execution times for these four versions are shown in figure 1.

Table 2 shows the precise numbers used to generate figure 1. It also includes the break-even point (in the Even column) and the time spent generating code (in the Gen column). Notice that the break-even points are less than 200, a relatively small number for numeric kernels like these that are likely to be used heavily.

3.2 Simulator

The SimpleScalar Toolset (Burger *et al.*, 1996) is a family of instruction-set simulators widely used in the hardware research community to model memory hierarchies. For these experiments, we ported the fastest functional simulator from the ToolSet into Cyclone and produced versions with and without RTCG. We then compared these two versions to the original C version compiled with `gcc -O2`. Our port was about 2300 lines of code (the RTCG port was only 20 lines longer).[2]

To fully evaluate the performance of our simulator we ran the SPEC CPU95 benchmark suite with modified inputs through our simulator.[3] The results for each

[2] Lines of code was measured with an `awk` script that excludes comments and blank lines.

[3] To limit the simulation time to about five hours, we used modified inputs commonly used with this simulator (Burger, 1998). This run is therefore not SPEC compliant.

https://doi.org/10.1017/S095679680200463X Published online by Cambridge University Press

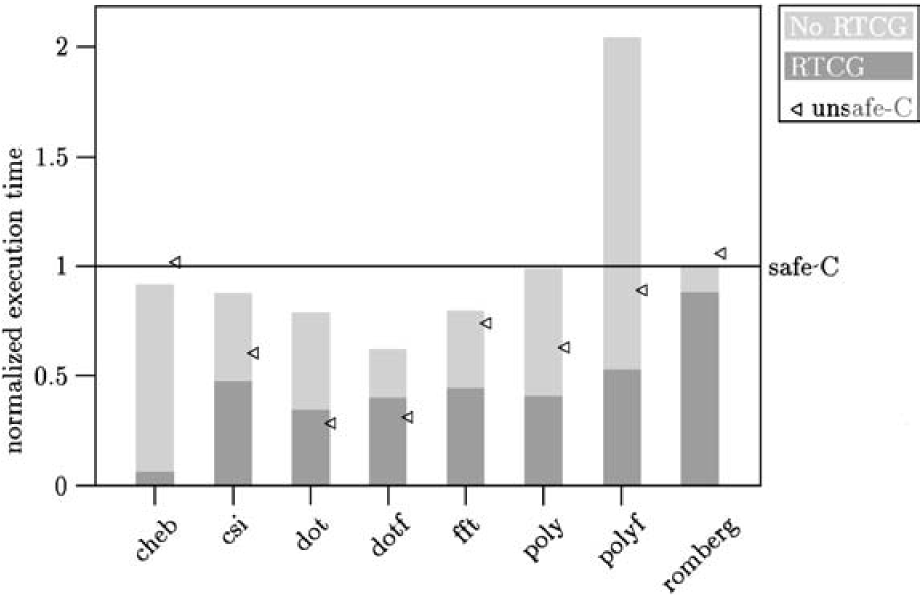

Fig. 1. Normalized execution times for several benchmark programs (excluding time for code generation). The RTCG and No-RTCG versions of each benchmark are compiled with our optimizing compiler. The unsafe-C and safe-C versions are compiled using `gcc -O2`. The safe-C version performs the same bounds check that the Cyclone versions do and is used as a baseline to normalize the execution times.

Table 2. *The execution times of the statically and dynamically compiled versions of these benchmarks. The Gen column reports the time used to dynamically generate code. The Even column shows the computed break-even point. All times are in cycles.*

Benchmark	unsafe-C	safe-C	no RTCG	RTCG	Gen	Even
cheb	83066	81785	75185	5171	379523	5
csi	16994	28232	24791	13421	12994	1
dot	193	683	538	238	26136	87
dotf	260	838	523	333	30724	162
fft	10655	14439	11587	6437	347501	67
poly	98	156	154	64	3216	36
polyf	10	170	193	393	4198	14
romberg	43779	41498	41263	36401	120134	25

SPEC CPU95 benchmark program are shown in figure 2. (Note that unlike the micro-benchmarks, these running times include the time for code generation.) The results show that on average the safe Cyclone program using RTCG runs 1.8 times faster than the optimized, but unsafe, C version. Many interesting detailed results are explained in the first author's dissertation (Smith, 2002).

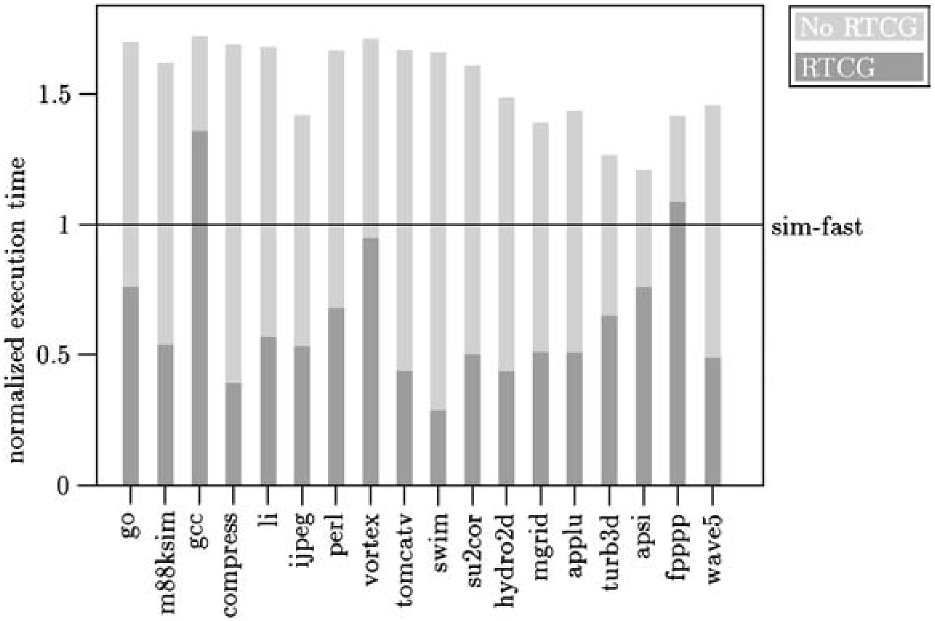

Fig. 2. Normalized execution times for SPEC CPU95 benchmark programs used as input to the simulator. The RTCG and No-RTCG versions of the simulator were compiled with our optimizing compiler. Execution times have been normalized so that the (unsafe) C version of the simulator (`sim-fast`) always takes 1.0 time.

4 Compilation overview

Cyclone is a type-safe C-like language supporting parametric polymorphism, exceptions, algebraic datatypes, and RTCG. For this paper, we restrict our examples to the C subset plus the four new RTCG constructs: `codegen`, `cut`, `splice`, and `fill`. These constructs are best conceptualized as manipulating “strings” to construct the text of a new function. For example, the function `dot_gen` below takes a vector `u` and returns a function specialized for taking dot products of `u`.

```
(int[] -> int) dot_gen(int u[]) {
  return codegen (
    int dot(int v[]) {
      int result = 0;
    cut { for(int i=0; i<size(u); i++)
      splice{ result += fill(u[i]) * v[fill(i)]; };
    }
    return result;});
}
```

The expression `codegen` (*int*(*dot*(*int v*[ ])...) indicates RTCG of a function. The generated function will be composed of copies of code fragments (templates) from

https://doi.org/10.1017/S095679680200463X Published online by Cambridge University Press

```
(int[] -> int) dot_gen_opt(int u[]) {
   return codegen (
      int dot(int v[]) {
        int result = 0;
        cut {
          for(int i=0; i<size(u); i++) {
            if(u[i]!=0) {
              if(u[i] == 1) splice { result += v[fill(i)]; }
              else splice { result += fill(u[i]) * v[fill(i)]; }
            }
          }
        }
      });
}
```

Fig. 3. A more aggressive version of dot_gen that specializes for 1 and 0. Dynamically generated code is shown in italics.

within the codegen expression. We have highlighted the templates by surrounding them with boxes; the boxes are not actually part of Cyclone.

The codegen expression takes an argument that is a function syntactically, but it is not executable. It contains templates, the first of which indicates the arguments and return type for the generated function. In our example, this template also includes the declaration of result. The code in the template is not executed during the invocation of dot_gen; instead, it is copied as the beginning of the generated function. The cut statement marks the end of the first template. The code within the cut is executed during the invocation of dot_gen. It iterates over the array u, executing a splice statement for each element. The splice statement marks the beginning of a new template; its argument is appended to the end of the function being generated. In our example, this argument contains fill expressions. These expressions mark *holes* in the template that are filled with values calculated by dot_gen; we have highlighted these holes with nested boxes. Finally, after the loop is finished executing, a copy of a third template ends the generated function. The result of the codegen is a pointer to the generated function. Invoking dot_gen on an array with elements {12,24,32} would thus generate a function like this:

```
int dot_generated(int v[]) {
  int result = 0;
  result += 12 * v[0];
  result += 24 * v[1];
  result += 32 * v[2];
  return result;
}
```

In this example, RTCG allows us to eliminate the looping overhead and to fold constants into the instructions. A more aggressive version, like the one shown in figure 3, could generate different code when an element of u is zero or one.

The RTCG primitives could be implemented using string manipulation and calls to a conventional compiler. Doing so would ensure that the resulting code was well

https://doi.org/10.1017/S095679680200463X Published online by Cambridge University Press

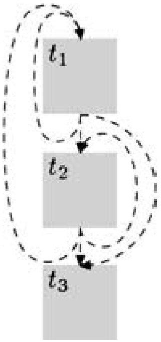

Fig. 4. All ways control might flow between `dot`'s templates. No edges flow out of t_3 because it ends in a `return` statement. The edges (t_2, t_1) and (t_1, t_1) are impossible only because of the way `dot_gen` assembles the templates.

optimized. For instance, we could reasonably expect that the `result` variable would be allocated a register. However, the overhead of a conventional compiler makes it difficult to take advantage of short-lived invariants.

We therefore use a different approach to generate code rapidly. As a first step, we translate dynamically generated functions into a collection of templates (code fragments with holes) in our internal representation. At compile-time, we perform register allocation of these templates to produce machine-code templates with holes.

These machine-code templates can be copied and their holes filled to produce a specialized function quickly. For example, the compiled `dot_gen` would use three templates: (t_1) for the function prologue and for initializing `result`, (t_2) for adding to `result` some constant value (`u[i]`) multiplied by a vector element at some constant offset (`i`), and (t_3) for returning `result` and the function epilogue.

To produce good code, we must perform optimizations on the templates, such as register allocation, at compile time. But how can we perform the necessary dataflow analyses, such as liveness analysis, when templates can be assembled dynamically at run time? The approach we suggest here is to perform dataflow analysis on the *generating* function to construct an approximation of the set of *control-flow graphs* of all functions that may be generated at run time. We compactly approximate this set as a single control-flow graph with two kinds of edges (see section 8). Standard control-flow optimizations can be extended to this representation.

For example, figure 4 depicts *all* ways control might flow between `dot`'s templates. By analyzing `dot_gen`, we discover that every specialized function it produces will have a restricted form: It will start with one copy of the first template, followed by some number of copies of the second template (appropriately specialized with constants), and then one copy of the third. In other words, the edges (t_2, t_1) and (t_1, t_1) in figure 4 could not be followed. Such an analysis would allow us to compute more accurate live ranges for the function's variables.

5 Mini-cyclone

Figure 5 presents the abstract syntax for Mini-Cyclone, a minimal subset of Cyclone that serves as this paper's source language. The primitive values are integers (i)

https://doi.org/10.1017/S095679680200463X Published online by Cambridge University Press

$$\begin{array}{llll}
\text{(integers)} & i & \in & \mathscr{Z} \\
\text{(function names)} & f & \in & \mathrm{VAR} \\
\text{(variables)} & x & \in & \mathrm{VAR} \\
\text{(expressions)} & e & ::= & i \mid f \mid x \mid e_1 + e_2 \mid e(e_1, \ldots, e_n) \\
 & & & \mid \texttt{codegen}\ D \mid \texttt{fill}\ e \\
\text{(statements)} & s & ::= & x := e \mid s_1 ; s_2 \mid \texttt{if}(e)\ s_1\ \texttt{else}\ s_2 \mid \texttt{while}(e)\ s \mid \texttt{return}\ e \\
 & & & \mid \texttt{cut}\ s \mid \texttt{splice}\ s \\
\text{(functions)} & D & ::= & f(x_1, \ldots, x_n)\ s \\
\text{(programs)} & P & ::= & D_1, \ldots, D_n
\end{array}$$

Fig. 5. Mini-cyclone syntax.

and function names (f). The other standard expressions are variables, addition, and function application. Similar to Cyclone, there are two expression forms for RTCG: `codegen` D generates a function and returns a pointer to it. `fill` e occurs in a generated function. Its meaning is to evaluate e as code in the function doing the code generation and embed the result, as a constant, in the generated function.

The standard statements (assignment, composition, conditional, loop, and return) have their usual meanings. The statement `cut` s must occur within a generated function (syntactically within a `codegen`). Its meaning is to execute s in the generating function at dynamic-code-generation time. The body of a `cut` statement can use a `splice` statement to append new code to the generated function. The statement `splice` s may occur only within a `cut` statement. Its meaning is to append the code computed from s onto the generated function. If s contains no `cut` statements then the computed code is simply s. Otherwise, each nested `cut` statement suspends generation of code from the `splice` statement, executes its argument, and then resumes generating code from the `splice` statement.

The RTCG constructs make sense only within certain contexts. For example, `cut` s and `splice` s may occur only within a `codegen` expression; and a `splice` makes sense only within a `cut`. The Cyclone type system, which has been shown sound (Hornof & Jim, 1999), ensures that these constructs are not misused. Because Mini-Cyclone is untyped, we simply assume that the input program is well-formed.

A Mini-Cyclone function takes a fixed number of arguments and executes a body. Syntactically, the argument to `codegen` is a function, but it is not executable.

6 Target language

Our compiler's intermediate representation is a standard low-level block-based language with special RTCG instructions and the type information needed to generate certified code. In this paper, we focus on compiling for RTCG, so we ignore typing issues and use a target language called CIR that is sufficient for compiling Mini-Cyclone.

Figure 6 shows the syntax for CIR. Reading from bottom to top:

- A CIR program (P) is a collection of functions (F).
- A function (F) has a list of parameters (x) and a list of blocks (B). The first

$$\begin{array}{lcl} i & \in & \mathscr{Z} \\ x & \in & \text{VAR} \\ l & \in & \text{LABEL} \\ h & \in & \text{HOLE} \\ t & \in & \text{TEMPLATE} \\ r & \in & \text{REGION} \\ p & \in & \text{INSTANCE} \end{array}$$

$$\begin{array}{ll} \text{(value)} & v ::= x \mid l \mid i \\ \text{(destination)} & d ::= l \mid \circ \mid [h] \end{array}$$

$$\begin{array}{lrl} \text{(instruction)} & \imath ::= & x := v \\ & \mid & x := v_1 + v_2 \\ & \mid & x := v(x_1, \ldots, x_n) \\ & \mid & x := [h] \\ & \mid & r := \underline{\texttt{start}}\ l \\ & \mid & p := \underline{\texttt{copy}}\ t\ \underline{\texttt{into}}\ r \\ & \mid & \underline{\texttt{fill}}\ p.[h]\ \underline{\texttt{with}}\ v \\ & \mid & \underline{\texttt{fill}}\ p_1.[h]\ \underline{\texttt{with}}\ p_2.l \\ & \mid & x := \underline{\texttt{end}}\ r \\ \text{(transfer)} & \chi ::= & \underline{\texttt{jmp}}\ d \mid \underline{\texttt{jnz}}\ x\,?\,d_1 : d_2 \mid \underline{\texttt{retn}}\ x \\ \text{(block)} & B ::= & (l, t, \imath_1 \ldots \imath_n, \chi) \\ \text{(function)} & F ::= & (x_1, \ldots, x_n) B_1, \ldots, B_m \\ \text{(program)} & P ::= & F_1, \ldots, F_n \end{array}$$

Fig. 6. CIR syntax.

block is the function's entry point; its label serves as the function's name. The order of blocks is significant.

- A (basic) block (B) has a label (l), a template name (t), an instruction sequence ($\imath_1, \ldots, \imath_n$), and a terminal control transfer (χ). Template names are used to group consecutive basic blocks. They are only meaningful in the containers used to dynamically generate code. (For ordinary functions, template names are ignored.)
- A control transfer (χ) is either a jump($\underline{\texttt{jmp}}\ d$), a conditional jump ($\underline{\texttt{jnz}}\ x\,?\,d_1 : d_2$), or a return($\underline{\texttt{retn}}\ \ x$). (The conditional jump ($\underline{\texttt{jnz}}\ x\,?\,d_1 : d_2$) jumps to d_1 if x is not zero, else it jumps to d_2.)
- An instruction ($\imath$) is either an assignment ($x := v$), an addition ($x := v_1 + v_2$), a function call ($x := v(x_1, \ldots, x_n)$), or a special purpose RTCG instruction. The RTCG-specific instructions are explained below.
- The destination for a control-transfer (d) is either the label for a block (l), a fall-through ($\circ$) or a hole($[h]$). A fall-through at the end of block B_i of a function is shorthand for the label B_{i+1}. Fall-throughs and holes will be useful for RTCG and are further discussed below.
- Values (v) are either variables (x), labels (l), or integers (i).

A CIR function represents either code that is directly executable or a *container* for templates. Executable functions must not contain holes, whereas containers can. The template names on the blocks within a container are used to group blocks into templates. Specifically, a template t is the list of blocks with template name t.[4] Executable functions use containers to generate code dynamically by *copying templates* (t) into *code regions* (r) and *filling* holes ($[h]$). We call a copy of a template an *instance* (p). Holes ($[h]$) allow generating code to specialize instances. Generating code can dictate the control flow of generated code by filling destination holes, and can insert values into the generated instructions by filling value holes ($x := [h]$).

[4] The order of blocks in the template is the same as their order in the function. Template names within executable functions are present only to simplify the syntax.

https://doi.org/10.1017/S095679680200463X Published online by Cambridge University Press

```
(u)  (dot_gen, _   r := start dot              (v)  (dot, t1,   result := 0,
                   p1 := copy t1 into r                         jmp ○)
                   i := 0,                          (l1, t2,    x := [h1]
                   jmp ○)                                       i := [h2]
     (l3, _        tst := i < size(u),                          tmp := x * v[i]
                   jnz tst ? l5 : ○)                            result := result + tmp,
     (l4, _        p2 := copy t2 into r                         jmp ○)
                   fill p2.[h1] with u[i]           (l2, t3,    , retn result)
                   fill p2.[h2] with i
                   i := i + 1,
                   jmp l3)
     (l5, _        p3 := copy t3 into r
                   f := end r,
                   retn f)
```

Fig. 7. Translation of the dot_gen example (simplified).

The instruction $r := \underline{\texttt{start}}\ l$ allocates a code region and binds it to r. All templates copied into r should come from the container function named l. The instruction $p := \underline{\texttt{copy}}\ t\ \underline{\texttt{into}}\ r$ puts an instance of t after the instances previously copied into r.[5] The term p becomes a reference to the new instance. The two fill instructions ($\underline{\texttt{fill}}\ p.[h]\ \underline{\texttt{with}}\ \ldots$) both fill the hole h of the specified instance (p). The second form allows one instance to refer directly to a label in another instance. That way, the generating code can create inter-template jumps in the generated code. Finally, $x := \underline{\texttt{end}}\ r$ completes the generation of a function and binds x to the resulting function pointer.

Inter-template jumps merit further discussion. It makes no sense for a template to refer directly to a label in another template. To which instance of a template would such a label refer? Therefore, the generating code must create all inter-template control flow using the $\underline{\texttt{fill}}\ p_1.[h]\ \underline{\texttt{with}}\ p_2.l$ instruction. There is an important exception: If a template's last block has a fall-through destination (such as $\underline{\texttt{jmp}}\ \circ$), then control will flow from an instance of that template to the next instance in the code region. That is, fall-through destinations refer to the code region's next block, not the container function's next block.

Figure 7 shows our dot_gen example in CIR (extended with arrays, etc.). The translation of section 7 would introduce more labels and variables but would otherwise produce similar output. Notice that the translation of *dot* contains holes and multiple templates, so it is not executable. When *dot_gen* is called, it creates a code region r in which to generate a function. It then copies *dot*'s prologue, t_1, and enters the loop consisting of blocks l_3 and l_4. The loop copies an instance of t_2 and binds p_2 to it. p_2 is used when filling holes h_1 and h_2. Its purpose is to specify which instance's holes to fill. Hole h_1 is filled with the value in $u[i]$ and hole h_2 is filled with the counter i. After the loop, *dot_gen* copies t_3 and ends the code generation.

[5] The implementation detects buffer overflow and moves the region to a larger buffer.

https://doi.org/10.1017/S095679680200463X Published online by Cambridge University Press

To employ inter-template optimizations on *dot*, it helps to have an approximation of the order its templates might be copied. Put another way, we seek to approximate the set of fall-through destinations t_1 and t_2 might refer to. Without looking at *dot_gen*, we must pessimistically assume that instances of t_1 and t_2 might be followed by instances of t_1, t_2, or t_3, but an analysis of *dot_gen* (see section 8) reveals that an instance of t_1 would never follow an instance of t_1 or t_2.

7 Translation

This section describes a translation from Mini-Cyclone (section 5) to CIR (section 6). We begin with an informal description of the subtle issues that RTCG raises and how we solve them. This overview highlights the main points that might otherwise remain hidden in the details of the translation. We then present our translation in two parts: We begin with the non-RTCG constructs and then integrate RTCG. This approach reflects our development of the actual Cyclone compiler and emphasizes one of our main conclusions, namely that we can integrate RTCG into a conventional optimizing compiler in a principled manner.

To structure our translation like a conventional compiler, the language for the translation (the meta-language) employs state and imperative features, such as an "emit" construct for generating target code. We find that using state makes the translation easier to understand and is better for showing that the translation is largely conventional. However, nothing precludes a more formal, purely functional translation. Appendix A presents a monadic-style translation that is actually quite similar to our stateful one.

Mini-Cyclone has no unstructured control flow (such as `break` or `goto`). At the end of this section, we explain how we can translate unstructured control flow into CIR, but the translation is less efficient than a richer target language could allow.

7.1 Highlights

If a Mini-Cyclone function named f contains `codegen` $g\ldots$, then we call f the *parent* and g the *child* because f generates a function using the templates contained in g. Cyclone has multilevel RTCG – a function generated from g might itself generate a function – so g might be the parent for some function named h. In the source language, children are lexically nested in parents, but CIR has no so such hierarchical structure.

Hence our one-pass translation must simultaneously generate multiple (top-level) CIR functions: When translating source code lexically contained in some number of function bodies, we may need to generate code into any of the functions' translations. To do so, the translation maintains an ordered sequence of environments, each of which includes a partially generated CIR function. The environment for a parent directly precedes the environment for a child. Therefore, the first environment in the sequence will become an executable CIR function whereas the others will become template containers.

One of the environments is designated *active*; instructions are emitted into its

https://doi.org/10.1017/S095679680200463X Published online by Cambridge University Press

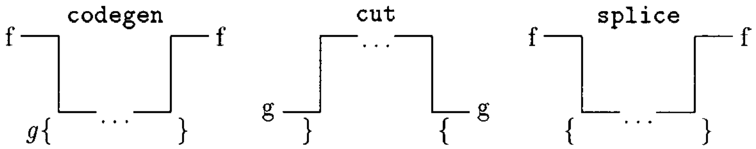

Fig. 8. During translation, these three RTCG constructs change the active function as shown. The higher line denotes the parent, and the lower line the child. The braces denote the start and end of templates.

CIR function. Translating `cut` makes the previous environment in the sequence (the environment for the parent) active. Dually, translating `splice` makes the next environment in the sequence (the environment for the child) active. Translating `codegen` creates a new environment, puts it after the active one, and makes the new environment active. Finally, to translate `fill` *e*, we make the parent's environment active to translate *e* and emit a CIR fill instruction. We then make the child's environment active again and emit a value-hole.

The translation also must partition the generated template containers into templates and emit copy instructions in parents. Clearly, the translations of distinct `splice` constructs must be in distinct CIR templates (i.e. in blocks with different template names). Otherwise, the parent would not have the flexibility to copy one statement a different number of times than another, such as happens when unrolling a loop. Hence the translation begins a new template in the child whenever the parent enters a `splice`. The translation also begins a new template in the child whenever the child leaves a `cut`. The child code after the `cut` is put in a new template, as opposed to the same template used before the `cut`, because otherwise the parent could not generate code in-between the child code surrounding the `cut`. Figure 8 summarizes the changes in the active environment and the extent of templates for `codegen`, `cut`, and `splice`.

Multiple templates create the possibility for inter-instance jumps. To see how such control flow arises, consider code of the form

```
while(e){ ...cut {...}...}...
```

The `cut` causes the code following the loop and the loop guard to be in different templates. Hence, the control transfer for a false guard is inter-instance. If the loop body had not contained a `cut`, then no inter-instance jump would have been necessary. Because our translation does only one pass over the syntax, it must decide whether the loop-guard transfer is inter-instance or intra-instance after translating the loop body. At that time, the translation checks if the template following the loop is the same as the template for the guard. If not, the translation places a destination hole in the guard and emits code (in the parent) to fill the hole (fill $p_1.[h]$ with $p_2.l$).

7.2 Non-RTCG translation

Each top-level function of the input program is translated independently, so to compile a program, the translation simply compiles each source function and collects

https://doi.org/10.1017/S095679680200463X Published online by Cambridge University Press

all the target functions created. If we ignore RTCG, our translation environment needs only the information necessary to translate a single function. For a language as simple as Mini-Cyclone without RTCG, the environment needs only a partially generated CIR function (into which we generate code) and a “current block,” the latter just being the label for a block in the function. An example use of the environment is the *emit* construct, which appends an instruction to the current block.

As described informally in section 7.1 and precisely in section 7.3, our translation environment is actually a sequence of function environments with a distinguished active environment. We use Φ to range over such a sequence. The adjustment for translating non-RTCG constructs is extremely simple: *emit* actually appends an instruction to the active environment’s current block.

We write $[[D]]$ for the translation of the function D. The role of $[[D]]$ is to translate the function’s body into a new function environment. Roughly, $[[f(x_1,\ldots,x_n)\ s]]$ creates a new function environment in which there is one empty block, makes the new environment active and the one block current, translates s, and finally adds the new environment’s now-completed CIR function to the output. We delay the precise definition until the next section because we use $[[D]]$ to translate `codegen`.

The translation of statements, $[[s]]$, takes a Φ as an argument. The translation of expressions, $[[e]]$, leaves its result in a CIR variable provided as input, so it takes a Φ and an x. The definitions $[[s]]$ and $[[e]]$ are inductive over the structure of Mini-Cyclone statements and expressions. The rest of this section presents the non-RTCG cases for these definitions.

All of the non-RTCG expression forms appear straightforward to translate. We use the primitive *newVar* to generate a fresh variable. The term $\mathit{emit}\ \Phi\ \iota$ adds ι to the end of the active environment’s current block. Similarly, $\mathit{emit}\ \Phi\ \chi$ makes χ the control transfer of the current block. $\mathscr{L}(f)$ is just the label for the first block in the translation of the function named f. Finally, $e_1;e_2$ means, “Do e_1 then e_2.”

$$\begin{array}{l} [[i]]\ \Phi\ x = \\ \quad \mathit{emit}\ \Phi\ x := i \\ \\ [[f]]\ \Phi\ x = \\ \quad \mathit{emit}\ \Phi\ x := \mathscr{L}(f) \\ \\ [[x]]\ \Phi\ y = \\ \quad \mathit{emit}\ \Phi\ y := x \end{array}$$

$$\begin{array}{l} [[e_1 + e_2]]\ \Phi\ x = \\ \quad \texttt{let}\ x_1 = \mathit{newVar}\ \texttt{in} \\ \quad \texttt{let}\ x_2 = \mathit{newVar}\ \texttt{in} \\ \quad [[e_1]]\ \Phi\ x_1; \\ \quad [[e_2]]\ \Phi\ x_2; \\ \quad \mathit{emit}\ \Phi\ x := x_1 + x_2 \end{array}$$

$$\begin{array}{l} [[e_0(e_1,\ldots,e_n)]]\ \Phi\ x = \\ \quad \texttt{let}\ x_0 = \mathit{newVar}\ \texttt{in} \\ \quad \ldots \\ \quad \texttt{let}\ x_n = \mathit{newVar}\ \texttt{in} \\ \quad [[e_0]]\ \Phi\ x_0; \\ \quad \ldots \\ \quad [[e_n]]\ \Phi\ x_n; \\ \quad \mathit{emit}\ \Phi\ x := x_0(x_1,\ldots,x_n) \end{array}$$

The following statement cases are just as straightforward:

$$\begin{array}{l} [[x := e]]\ \Phi = \\ \quad [[e]]\ \Phi\ x \end{array}$$

$$\begin{array}{l} [[s_1;s_2]]\ \Phi = \\ \quad [[s_1]]\ \Phi; \\ \quad [[s_2]]\ \Phi \end{array}$$

$$\begin{array}{l} [[\texttt{return}\ e]]\ \Phi = \\ \quad \texttt{let}\ x = \mathit{newVar}\ \texttt{in} \\ \quad [[e]]\ \Phi\ x; \\ \quad \mathit{emit}\ \Phi\ \underline{\texttt{retn}}\ x \end{array}$$

The control-flow constructs (`if` and `while`) are more complicated. Both definitions follow the same strategy: Generate the subterms while remembering the entry

https://doi.org/10.1017/S095679680200463X Published online by Cambridge University Press

and exit blocks for each. Then replace various transfers to create the correct control flow. As explained earlier, the translation may have to create inter-instance control flow at these points. We use the term *genDest* for this purpose. In the absence of RTCG, $\mathit{genDest}\ \Phi\ (b_{src}, b_{dst}) = b_{dst}$; the next section gives the complete definition.

The term *changeBlock* puts a new block in the active environment and makes the new block the current one. *changeBlock* returns a pair of the old current block's label and the new current block's label. The new block has the same template name as the old block. The term $\mathit{emit}\ \Phi\ \ b{:}\ \chi$ replaces the transfer in block b with χ.

```
$[[\texttt{if}(e)\ s_1\ \texttt{else}\ s_2]]\ \Phi =$
    $\texttt{let}\ x = \mathit{newVar}\ \texttt{in}$
    $[[e]]\ \Phi\ x;$
    $\texttt{let}\ (b_0, b_t) = \mathit{changeBlock}\ \Phi\ \texttt{in}$
    $[[s_1]]\ \Phi;$
    $\texttt{let}\ (b_1, b_f) = \mathit{changeBlock}\ \Phi\ \texttt{in}$
    $[[s_2]]\ \Phi;$
    $\texttt{let}\ (b_2, b_m) = \mathit{changeBlock}\ \Phi\ \texttt{in}$
    $\texttt{let}\ d_f = \mathit{genDest}\ \Phi\ (b_0, b_f)\ \texttt{in}$
    $\texttt{let}\ d_m = \mathit{genDest}\ \Phi\ (b_1, b_m)\ \texttt{in}$
    $\mathit{emit}\ \Phi\ \ b_0{:}\ \underline{\texttt{jnz}}\ x\ ?\circ : d_f;$
    $\mathit{emit}\ \Phi\ \ b_1{:}\ \underline{\texttt{jmp}}\ d_m;$
    $\mathit{emit}\ \Phi\ \ b_2{:}\ \underline{\texttt{jmp}}\ \circ$
```

```
$[[\texttt{while}(e)\ s]]\ \Phi =$
    $\texttt{let}\ (b_0, b_t) = \mathit{changeBlock}\ \Phi\ \texttt{in}$
    $\texttt{let}\ x = \mathit{newVar}\ \texttt{in}$
    $[[e]]\ \Phi\ x;$
    $\texttt{let}\ (b_1, b_b) = \mathit{changeBlock}\ \Phi\ \texttt{in}$
    $[[s]]\ \Phi;$
    $\texttt{let}\ (b_2, b_e) = \mathit{changeBlock}\ \Phi\ \texttt{in}$
    $\texttt{let}\ d_e = \mathit{genDest}\ \Phi\ (b_1, b_e)\ \texttt{in}$
    $\texttt{let}\ d_t = \mathit{genDest}\ \Phi\ (b_2, b_t)\ \texttt{in}$
    $\mathit{emit}\ \Phi\ \ b_0{:}\ \underline{\texttt{jmp}}\ \circ;$
    $\mathit{emit}\ \Phi\ \ b_1{:}\ \underline{\texttt{jnz}}\ x\ ?\circ : d_e;$
    $\mathit{emit}\ \Phi\ \ b_2{:}\ \underline{\texttt{jmp}}\ d_t$
```

7.3 RTCG translation

To precisely explain the translation of the RTCG constructs, we must be more exact in our treatment of environments. An environment has (translation-language) type τ_Φ, which we define as follows:

$$\begin{aligned}\tau_\Phi &= \{\mathbf{parents} : \tau_{\mathscr{E}}\ \text{list},\ \mathbf{active} : \tau_{\mathscr{E}},\ \mathbf{children} : \tau_{\mathscr{E}}\ \text{list}\}\\ \tau_{\mathscr{E}} &= \{\mathbf{current} : \textsc{label},\ \mathbf{fun} : \text{function}\ \}\end{aligned}$$

We define important auxiliary functions for changing a sequence's active environment:

$$\begin{aligned}\mathit{child}(\Phi) = \{&\mathbf{parents} = \Phi.\mathbf{active} :: \Phi.\mathbf{parents}, \mathbf{active} = \mathit{head}(\Phi.\mathbf{children}),\\ &\mathbf{children} = \mathit{tail}(\Phi.\mathbf{children})\}\\ \mathit{parent}(\Phi) = \{&\mathbf{children} = \Phi.\mathbf{active} :: \Phi.\mathbf{children}, \mathbf{active} = \mathit{head}(\Phi.\mathbf{parents}),\\ &\mathbf{parents} = \mathit{tail}(\Phi.\mathbf{parents})\}\end{aligned}$$

So $\mathit{child}(\Phi)$ makes the child active and $\mathit{parent}(\Phi)$ makes the parent active. Translation of well-formed source code never applies *head* or *tail* to an empty list. (In the implementation, this property is guaranteed by the Cyclone type-checker.)

We have already used *newVar* to generate fresh CIR variables. We also need to generate fresh template names, region names, holes, and instance names. Furthermore, we need to relate members of one of these syntactic classes to another. For

https://doi.org/10.1017/S095679680200463X Published online by Cambridge University Press

example, when emitting a fill for a hole in an instance of a template named t, we need to know the name that the parent uses for the instance.

Rather than keeping an explicit correspondence between members of different syntax classes, we assume the classes are isomorphic and that we have functions witnessing the isomorphisms: Given an element of any class (call it a), we can generate the corresponding variable ($\mathscr{V}(a)$), label ($\mathscr{L}(a)$), hole ($\mathscr{H}(a)$), template name ($\mathscr{T}(a)$), region name ($\mathscr{R}(a)$), or instance name ($\mathscr{P}(a)$). This technical trick lets us exploit subtle invariants. For example, the translation has the property that a template's instances are manipulated by the parent one at a time, so we can use the template name (t) to induce an instance name ($\mathscr{P}(t)$). It also lets *newVar* suffice for creating fresh members of any class. For example, $\mathscr{H}(\mathit{newVar})$ is a new hole.

Given these preliminary considerations, we can present the translation of `codegen`:

$$\begin{array}{l}
[[\texttt{codegen}\ f(x_1,\ldots,x_n)s]]\ \Phi\ x = \\
\quad \mathit{emit}\ \Phi\ \mathscr{R}(f) := \underline{\texttt{start}}\ \mathscr{L}(f); \\
\quad \mathit{emit}\ \Phi\ \mathscr{P}(f) := \underline{\texttt{copy}}\ \mathscr{T}(f)\ \underline{\texttt{into}}\ \mathscr{R}(f); \\
\quad [[f(x_1,\ldots,x_n)s]]\ \Phi; \\
\quad \mathit{emit}\ \Phi\ x := \underline{\texttt{end}}\ \mathscr{R}(f)
\end{array}$$

The translation emits parent code and child code. In the parent, we allocate the code region ($\mathscr{R}(f) := \underline{\texttt{start}}\ \mathscr{L}(f)$) and copy the child's first template ($\mathscr{P}(f) := \underline{\texttt{copy}}\ \mathscr{T}(f)\ \underline{\texttt{into}}\ \mathscr{R}(f)$). For translating the child, we simply call $[[f(x_1,\ldots,x_n)s]]$ with the current environment. That translation function (defined below) will make the active environment the parent of the newly generated function. That way, `cut` statements in s will use the correct local environment. After the child has been translated, the parent ($\Phi.\textbf{active}$) will not have any more instructions emitted into it that manipulate this child. The last step is to emit code to convert the code region into executable code ($x := \underline{\texttt{end}}\ \mathscr{R}(f)$).

Our use of $[[D]]$ essentially implies its full definition:

$$\begin{array}{l}
[[f(x_1,\ldots,x_n)\ s]]\ \Phi = \\
\quad \texttt{let}\ \mathscr{E} = \{\textbf{current} = \mathscr{L}(f), \textbf{fun} = (x_1,\ldots,x_n)(\mathscr{L}(f), \mathscr{T}(f), , \underline{\texttt{jmp}}\ \circ)\}\ \texttt{in} \\
\quad [[s]]\ \Phi[\textbf{parents} = \Phi.\textbf{active} :: \Phi.\textbf{parents}, \textbf{active} = \mathscr{E}, \textbf{children} = \emptyset]; \\
\quad \mathit{addFun}\ \mathscr{E}
\end{array}$$

(The notation $\Phi[\textbf{label} = F]$ is functional record update; the result has the same fields as Φ except field **label** has value F.) We create a new function environment with one empty block having label $\mathscr{L}(f)$ and template name $\mathscr{T}(f)$. (The translation of s will replace the block's transfer.) We use *addFun* to add the completed function to the CIR program. The essential point is that $\Phi[\textbf{parents} = \Phi.\textbf{active} :: \Phi.\textbf{parents}, \textbf{active} = \mathscr{E}, \textbf{children} = \emptyset]$ ensures that the translation of s takes place in the proper environment with the correct parent. (Although we have cleared the field **children** to emphasize that the values it contains will not be used by $[[s]]$, this step is unnecessary as the Cyclone type-checker enforces this property.)

The translations for `cut` and `splice` are pleasingly symmetric:

$$\begin{array}{l}
[[\texttt{cut}\,s]]\ \Phi = \\
\qquad emit\ \ \Phi\ \underline{\texttt{jmp}}\ \circ; \\
\qquad [[s]]\ parent(\Phi); \\
\qquad newTemplate\ \ \Phi \\
\\
[[\texttt{splice}\,s]]\ \Phi = \\
\qquad newTemplate\ \ child(\Phi); \\
\qquad [[s]]\ child(\Phi); \\
\qquad emit\ \ child(\Phi)\ \underline{\texttt{jmp}}\ \circ
\end{array}
\qquad
\begin{array}{l}
newTemplate\ \ \Phi = \\
\qquad \texttt{let}\ f = activeFunction\ \ \Phi\ \texttt{in} \\
\qquad \texttt{let}\ x = newVar\ \texttt{in} \\
\qquad \texttt{let}\ (b_0, b_1) = changeBlock\ \ \Phi\ \texttt{in} \\
\qquad setTemplateOfCurrent\ \ \Phi\ \mathscr{T}(x); \\
\qquad emit\ \ parent(\Phi)\ \mathscr{P}(x) := \underline{\texttt{copy}}\ \mathscr{T}(x)\ \underline{\texttt{into}}\ \mathscr{R}(f)
\end{array}$$

Recall that the purpose of `cut` s is to execute s in the parent of the function in which the `cut` appears. Dually, the argument to `splice` is part of the child function. In both cases, we have a child-to-parent transition (beginning of the cut, end of the splice) and a parent-to-child transition (end of the cut, beginning of the splice). For child-to-parent transitions, we end the current block so that control will flow to a new template that an ensuing parent-to-child transition creates. For parent-to-child transitions, we add a new template because the parent may choose to copy the code after the transition at different times than the code before the transition.

The auxiliary term *newTemplate* creates a new template in the child and a copy instruction in the parent. We use a fresh template name and use it to induce a fresh instance name. Because of the translation of `codegen`, $\mathscr{R}(f)$ is the correct region name for the copy. (*activeFunction* Φ retrieves the active function's name. *setTemplateOfCurrent* $\Phi\ t$ changes the current block's template name to t.)

The translation of `fill` e is straightforward at this point. In the parent, we translate e and emit a $\underline{\texttt{fill}}$ instruction for a new hole. In the child, we emit a value hole. Because of *newTemplate*, the correct instance name for the $\underline{\texttt{fill}}$ is $\mathscr{P}(t)$. (*currentTemplate* retrieves the current block's template name.)

$$\begin{array}{l}
[[\texttt{fill}\ e]]\ \Phi\ x = \\
\qquad \texttt{let}\ x_1 = newVar\ \texttt{in} \\
\qquad [[e]]\ (parent(\Phi))\ x_1; \\
\qquad \texttt{let}\ x_2 = newVar\ \texttt{in} \\
\qquad \texttt{let}\ t = currentTemplate\ \ \Phi\ \texttt{in} \\
\qquad emit\ \ (parent(\Phi))\ \underline{\texttt{fill}}\ \mathscr{P}(t).[\mathscr{H}(x_2)]\ \underline{\texttt{with}}\ x_1; \\
\qquad emit\ \ \Phi\ x := [\mathscr{H}(x_2)]
\end{array}$$

So far, the translation does not seem to use any $\underline{\texttt{fill}}\ p_1.[h]\ \underline{\texttt{with}}\ p_2.l$ instructions. As discussed previously, we need such instructions for inter-instance jumps. We have just seen that new templates are created when `cut` statements occur inside of other source constructs. This fact is why the control-flow jumps in the translation of `if` and `while` may be inter-instance. We now have the machinery to define *genDest* in general: We simply check whether the templates of the source and destination blocks are the same (using *templateOf* $\Phi\ b$ to retrieve the template name in the block with label b). If the same, we just return the destination's label. If different, we create a hole, emit a fill for the hole in the parent, and return the hole.

```
$genDest\ \Phi\ (b_{src}, b_{dst}) =$
    let $t_{src} = templateOf\ \Phi\ b_{src}$ in
    let $t_{dst} = templateOf\ \Phi\ b_{dst}$ in
    if $t_{src} = t_{dst}$ then $b_{dst}$
    else (let $x = newVar$ in
        $emit\ (parent(\Phi))$ $\underline{\texttt{fill}}$ $\mathscr{P}(t_{src}).[\mathscr{H}(x)]$ $\underline{\texttt{with}}$ $\mathscr{P}(t_{dst}).b_{dst}$;
        $[\mathscr{H}(x)]$)
```

7.4 *Unstructured control flow*

Unlike Cyclone, Mini-Cyclone does not have unstructured jumps, such as `break` or `goto`. Like the "forward jumps" in the translation of conditionals and loops, such constructs can lead to inter-instance jumps, which require jump holes and code to fill them. Unfortunately, with unstructured code, we cannot statically bound the number fills needed. For example, in the following program, the function `f` will generate `x` jumps to *s*, but our translation cannot fill the holes for these jumps until the body of the child's loop is translated.

```
f(x)
 return codegen g() {
   while(e) {
     cut { while(x>0) {
             splice { ... break ... };
             x = x-1; } } };
   s }
```

We know two solutions to this problem. First, we could enrich CIR with data structures and first-class holes. Then the translation could have the parent maintain a list of jump-holes to fill after generating the child's loop. One potential disadvantage of this solution is that it becomes much harder for a dataflow analysis to accurately determine the destinations with which the parent might fill these holes.

The second solution turns forward jumps into backward jumps by generating a single block before the child's loop with a jump-hole. We translate `break` to jump to this block and later fill the one hole with the necessary destination. The disadvantage is efficiency: Unstructured jumps are translated into two jumps.

8 Analysis and optimization

This section shows how to perform intra-procedural dataflow optimizations over CIR functions. A preliminary step in dataflow analyses is to build a control-flow graph (CFG). A CFG for a function F is a directed graph whose nodes are the labels in F. The graph must have an edge from l_1 to l_2 if control may flow directly from the end of block l_1 to the beginning of block l_2.

This definition of CFG does not apply to template container functions because there is no control flow in the containers *per se*. Rather, we are interested in how

```
gen_f(n) {
  return codegen(
     f(m,max) {
       ans = 0;
       while(ans<max) {
         y   = fill(n) * m;
         ans = y + ans;
       }
       return ans;
       });
}
```

```
gen_g(n) {
  return codegen(
     g(m,max) {
       ans = 0;
       cut {
         do {
           splice {
             if(max > ans) {
               y   = fill(n) * m;
               ans = y + ans;
             }
           };
           n = n - 1;
         } while(n > 0);
       }
       return ans;
       });
}
```

Fig. 9. Cyclone code used to demonstrate why we distinguish inter-template and intra-template edges. Instances of f return the least multiple of $n * m$ greater than max. Instances of g compute $\Sigma_{i=1}^{n} i * m$ in descending order, stopping if a partial sum exceeds max.

control may flow in the functions *generated* from the container. We want the CFG for a container function to approximate control flow for all functions that may eventually be generated from it. We therefore generalize the definition of CFG as follows: A CFG for a container function F is a directed graph, with inter-template and intra-template edges, whose nodes are the labels in F and where control may flow from an *instance* of l_1 to an *instance* of l_2 only if there is an edge of some kind from l_1 to l_2. The two kinds of edges represent different degrees of precision:

- An inter-template edge between (l_1, l_2) allows control to flow from any instance of l_1 to any instance of l_2.
- An intra-template edge between (l_1, l_2) allows control to flow only between copies of l_1 and l_2 that occur in the same instance.

8.1 Examples

To understand our decision to distinguish intra-template and inter-template edges, consider the two (slightly embellished) Mini-Cyclone functions in figure 9. The behaviors of f and g are not as important as the similarity of their (simplified) translations shown in figure 10.

The significant difference between these two container functions is that the translation of the loop in f has an intra-template back-edge (from f_2 to f_1) whereas g has a cycle caused by an inter-template edge (from g_3 to g_1). Despite this difference, the control-flow graphs (see figure 11) look quite similar if we ignore the edge kinds. Yet f and g are used very differently: functions generated from f have one copy of each block in the order they appear in f. Functions generated from g are loop-free and contain n instances of t_1.

https://doi.org/10.1017/S095679680200463X Published online by Cambridge University Press

$$\begin{array}{ll}
(m, max) & (f, t,\ ans := 0, \\
 & \qquad \underline{\texttt{jmp}}\ \circ) \\
 & (f_1, t,\ tst := max > ans, \\
 & \qquad \underline{\texttt{jnz}}\ tst\ ?\circ : f_3) \\
 & (f_2, t,\ x := [h], \\
 & \qquad y = x * m, \\
 & \qquad ans = y + ans, \\
 & \qquad \underline{\texttt{jmp}}\ f_1) \\
 & (f_3, t,\ \underline{\texttt{retn}}\ ans)
\end{array}
\qquad
\begin{array}{ll}
(m, max) & (g, t_0,\ ans := 0, \\
 & \qquad \underline{\texttt{jmp}}\ \circ) \\
 & (g_1, t_1,\ tst := max > ans, \\
 & \qquad \underline{\texttt{jnz}}\ tst\ ?\circ : g_3) \\
 & (g_2, t_1,\ x := [h], \\
 & \qquad y = x * m, \\
 & \qquad ans = y + ans, \\
 & \qquad \underline{\texttt{jmp}}\ \circ) \\
 & (g_3, t_1,\ \underline{\texttt{jmp}}\ \circ) \\
 & (g_4, t_2,\ \underline{\texttt{retn}}\ ans)
\end{array}$$

Fig. 10. Simplified translations into CIR of the container functions from figure 9.

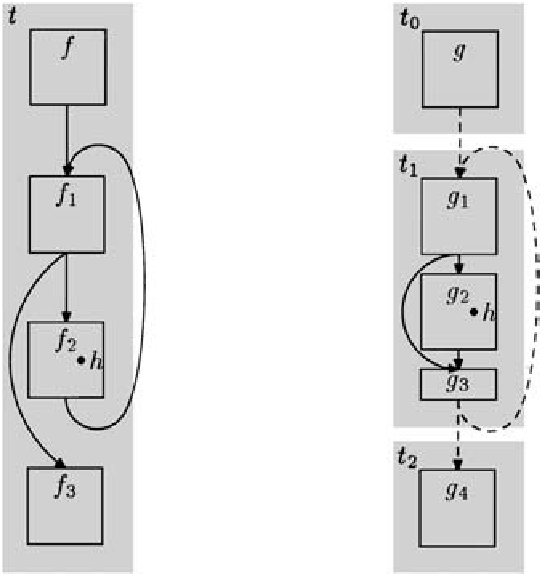

Fig. 11. Control-flow graphs for f and g reflect differences in their generators. Solid arrows denote intra-template edges; dashed arrows denote inter-template edges.

The CFGs capture most, but not all, of this information. The graph for f tells us that the loop is intra-template, but we do not know how many times t may be copied.[6] From the CFG for g, we must conservatively assume that the inter-template edge from g_3 to g_1 might give rise to an intra-instance edge or an inter-instance edge in a generated function. In this sense, inter-template edges represent a proper superset of the possibilities that intra-template edges represent. However, we have two ways to reason that there will not be an intra-instance edge from g_3 to g_1. First, only hole destinations can give rise to intra-instance control flow and g_2 has a fall-through destination. Secondly, the translation in section 7 never fills a hole with a label from the same template.

[6] In this example, it happens that subsequent copies would be unreachable.

https://doi.org/10.1017/S095679680200463X Published online by Cambridge University Press

The most important difference in the CFGs for f and g is that the loop in g has an inter-template edge. In the next section, we will demonstrate this difference's relevance (and often, its irrelevance) to dataflow analysis.

8.2 Dataflow analysis over CIR functions

The reason we have distinguished inter-template edges is that holes are constant in intra-template loops, but not necessarily in inter-template loops. After all, a loop containing an inter-template edge can lead to a generated function with multiple instances of a template, each with its holes filled with *different* constants.

Analyses not concerned with the constancy of holes need not differentiate between the kinds of edges. For example, consider liveness analysis, which is crucial to inter-template register allocation in our compiler. Because the uses and definitions of variables occur in the same place in every template instance, it does not help to distinguish control flow to the same instance (i.e., the intra-template edges). Indeed, the variables live after each corresponding instruction in our example functions are the same (although in other examples the edge (g_3, g_4) could change the result).

On the other hand, distinguishing edges allows an analysis to determine that more expressions are loop invariant. Given a loop in a conventional CFG, we say an expression in the loop is invariant if it must have the same value from the time code in the loop is entered until code not in the loop is entered. With RTCG, we mean that in all generated functions, all instances of an expression must have the same value – and the same value as all other instances – from the time an instance of code in the loop is entered until an instance of code not in the loop is entered. Other definitions of invariance are less useful for optimization. For example, the expression $x * m$ is invariant in f_2, but not in g_2. An optimizing compiler could hoist the first two instructions from f_2 to f, but hoisting the same code from g_2 to g would not preserve meaning.[7]

We cannot speak precisely about an analysis being sound without defining a formal semantics for CIR. Informally, a container function represents the set of functions that might be generated from it. An analysis is sound when its results are sound for all functions in this set. Because these functions have no inter-template edges, established soundness results apply.

8.3 Transformations over CIR functions

One standard use of dataflow analysis is to direct meaning-preserving code transformations. In a conventional language, an intra-procedural transformation has no effect on any other function, so long as the function's interface is unchanged. With RTCG, a container function's "interface" to the rest of the program includes how its parent copies templates and fills holes. In other words, any change to the function's templates or holes may affect how its parent needs to generate the function.

[7] We could hoist the instructions in g_2 to a new loop-header block in t_1, but doing so provides no benefit and increases register pressure in g_1.

https://doi.org/10.1017/S095679680200463X Published online by Cambridge University Press

The changes in a parent that intra-template transformations require are straightforward. Essentially, a hole might be deleted (e.g. dead-code elimination), duplicated (e.g. copy propagation), or moved (e.g. instruction scheduling). It is easy to find all fills of the hole in a parent and do the corresponding deletion or duplication. (Intra-template hole motion requires no change in a parent.) Note that the soundness of an intra-template transformation may rely on inter-template information. For example, suppose our example function f had an additional template that assigned 0 to y and then had an inter-template jump to f_2. In this case, we could not move the hole and multiplication out of the loop.

Inter-template hole transformations are more difficult. For example, suppose it is sound to move a hole from a template t to a different template t'. Doing so requires rewriting the parent's fill instruction to refer to the appropriate instance of t', but that instance pointer may not be available. Therefore, our compiler does not attempt inter-template hole motion. In the future, we hope to find a convenient way to communicate the necessary information between the parent and child to permit such transformations.

The Cyclone compiler performs some simple template transformations. Unreachable templates are eliminated as are "empty templates" (templates equivalent to a single fall-through). In both cases, it suffices to remove from the parent all copy instructions that mention the empty template.

Note that a sound transformation of a parent cannot require changing a child: Because the parent treats the child as data, any correct optimization will copy the templates in the same order and fill the holes with the same values.

The Cyclone compiler implements register allocation, liveness analysis, inlining, and null-check elimination uniformly over functions and template container functions. The same code is used for these two kinds of templates with very little modification. As discussed, these transformations and analyses are sound because they do not involve code motion.

8.4 Control-flow graphs for CIR functions

This section describes the analysis used in the Cyclone compiler to compute generalized CFGs for container functions. Prior to implementing this analysis our compiler tried to construct the CFG during translation. We found that approach difficult to maintain and less precise than this analysis. Minimizing edges in the CFG is important because it affects the precision of all subsequent analyses. Furthermore this analysis can be run after any number of other analyses to reconstruct the CFG should that be necessary.

There are two distinct kinds of edges in the CFG for F, intra-template edges (`intra`(F)) and inter-template edges (`inter`(F)). The edges in `intra`(F) represent control flow within a template instance. They arise from label destinations and from intra-template fall-through destinations. The edges in `inter`(F) represent control flow that may be to another instance.[8] They arise from hole destinations

[8] The name is misleading because the control flow could be to the same instance.

https://doi.org/10.1017/S095679680200463X Published online by Cambridge University Press

and fall-through destinations in a template's last block. These edges approximate how the generator(s) of F might create control flow by filling holes and copying templates.

The computation of $\texttt{intra}(F)$ is standard: Let $F = (x_1, \dots, x_n)B_1, \dots, B_m$ and $B_i = (l_i, t_i, \dots, \chi_i)$. Let d be a destination in χ_i. If $d = l_j$ and $t_i = t_j$, then $(l_i, l_j) \in \texttt{intra}(F)$.[9] If $d = \circ$, $i < m$, and $t_i = t_{i+1}$, then $(l_i, l_{i+1}) \in \texttt{intra}(F)$. No other edges are in $\texttt{intra}(F)$.

Computing $\texttt{inter}(F)$ is more interesting. Given only F, we could compute only a very conservative $\texttt{inter}(F)$: Any hole could be filled with any label of any instance and an instance of any template could follow an instance of any other. By analyzing F's parents (that is, functions containing $r := \underline{\texttt{start}}\ f$ where f is F's name), we can attain more precision. Our translation has the property that executable functions have no parents and non-executable functions have exactly one parent. We use $\mathit{parent}(F)$ to denote the parent of F. We explain how to extend the analysis to multiple parents at the end of the section.

The Cyclone compiler uses a forward dataflow analysis on $\mathit{parent}(F)$ to compute $\texttt{inter}(F)$. The compiler first computes the parent's CFG and then uses it to compute the child's CFG. (If F has no parent, then $\texttt{inter}(F)$ has no edges.)

As discussed earlier, because this analysis does not depend on the values of holes, it need not distinguish between the two kinds of edges. For now, assume that $\mathit{parent}(F)$ contains exactly one $r := \underline{\texttt{start}}\ f$ instruction. Our goal is to determine what edges we must add to the CFG for F because of the way $\mathit{parent}(F)$ manipulates the templates. We assume every instruction in $\mathit{parent}(F)$ is reachable, so the edges added are the union of the edges each instruction adds:

$$\texttt{inter}(F) = \bigcup_{\imath \in \mathit{parent}(F)} \texttt{inter}(\imath)$$

Instructions add edges by filling holes and copying templates:

$$\texttt{inter}(\imath) = \begin{cases} \{(l_i, l_j)\} & \text{if } \imath = \underline{\texttt{fill}}\ p_1.[h]\ \underline{\texttt{with}}\ p_2.l_j \text{ and } h \text{ is in } \chi_i \\ \texttt{copypred}(\imath) \times \{l\} & \text{if } \imath \text{ copies } t \text{ and } t\text{'s first block has label } l \\ \emptyset & \text{otherwise} \end{cases}$$

The first case includes edges introduced by filling holes. The second case is for fall-through destinations: If $\imath$ is a copy instruction that copies t, then consider all t' that could have been the most recently copied template when control reaches $\imath$. If the last block of t' has a fall-through destination, then we must add an edge from the last block of t' to the first block of t. We write $\texttt{copypred}(\imath)$ for the set of labels of such "last blocks". Determining $\texttt{copypred}(\imath)$ for each $\imath$ in $\mathit{parent}(F)$ is a dataflow problem. We define the function $\texttt{fallthru}(t)$ for a template in F to be the label of the last block if that block has a fall-through destination:

$$\texttt{fallthru}(t) = \begin{cases} \{l\} & \text{if } (l, t, \dots, \chi) \text{ is } t\text{'s last block and } \chi \text{ contains } \circ \\ \emptyset & \text{otherwise} \end{cases}$$

[9] If $d = l_j$ and $t_i \neq t_j$, then F is not well-formed.

https://doi.org/10.1017/S095679680200463X Published online by Cambridge University Press

For an instruction sequence in *parent*(F), `last` denotes the last template copied, if any:

$$\begin{array}{rcll} \texttt{last}(\imath_1,\ldots,\imath_n) & = & \left\{\begin{array}{c} \{t\} \\ \texttt{last}(\imath_1,\ldots,\imath_{n-1}) \end{array}\right. & \begin{array}{l} \text{if } \imath_n = p := \underline{\texttt{copy}}\ t\ \underline{\texttt{into}}\ r \\ \text{otherwise} \end{array} \\ \texttt{last}(\cdot) & = & \emptyset & \end{array}$$

We extend `last` to blocks via $\texttt{last}(l,t,\imath_1,\ldots,\imath_n,\chi) = \texttt{last}(\imath_1,\ldots,\imath_n)$. We can now state the relevant dataflow equations.

$$\begin{array}{rcll} out(B) & = & \left\{\begin{array}{c} \texttt{fallthru}(t) \\ in(B) \end{array}\right. & \begin{array}{l} \text{if } \texttt{last}(B) = \{t\} \\ \text{if } \texttt{last}(B) = \emptyset \end{array} \\ in(B) & = & \displaystyle\bigcup_{B' \in \texttt{pred}(B)} out(B') & \end{array}$$

These equations specify a forward dataflow problem that can be solved using standard techniques (Aho *et al.*, 1986; Muchnick, 1997), starting with $in(B) = \emptyset$ for the first block of *parent*(F). Note that we use the assumption that CFG for *parent*(F) is already defined in the definition of $in(B)$. Using the dataflow solution and letting $\imath_{(l,i)}$ denote the i^{th} instruction in block l, we can define `copypred`:

$$\texttt{copypred}(\imath_{(l,i)}) = \left\{\begin{array}{cl} in(B) & \text{if } i = 1 \text{ and } B = (l,\ldots) \\ \texttt{fallthru}(t) & \text{if } \imath_{(l,i-1)} = p := \underline{\texttt{copy}}\ t\ \underline{\texttt{into}}\ r \\ \texttt{copypred}(\imath_{(l,i-1)}) & \text{otherwise} \end{array}\right.$$

The analysis we have defined above is adequate for the subset of CIR produced by our translation. It does not, however, handle multiple parents nor mutually generating functions. Although the utility of these features is unclear to us, we speculate briefly on how to generalize our analysis to handle them. If F has multiple parents, we just add the edges that each parent requires. Similarly, if one parent uses multiple code regions, we just repeat the process for each code region. Because code regions and instance pointers are second-class constructs, we always know which code region an instruction affects. The interesting extension is for mutually generating functions. That is, nothing in CIR prevents a cycle in the graph where edge (F,G) means F is a parent for G. Here we cannot compute the CFG for a parent before a child. The solution is to iterate the program-wide analysis in a manner just like conventional inter-procedural analysis.

Handling more expressive languages like 'C would require more generalization. First consider a system in which multiple functions coordinate to generate code. In this case, we could either fall back on a conservative approximation for inter-template control-flow at procedure boundaries or use an inter-procedural analysis of the coordinating parent functions. We could further allow a single template to belong to more than one container function. Keeping transformations consistent across all the container functions using a particular template would be challenging. This situation is somewhat analogous to inter-procedural optimization where the templates are the procedures and the container functions are the exposed interface. Both of these generalizations would be useful additions to Cyclone, but they would likely lead to less precise CFGs and therefore less effective static optimizations.

9 Future work

We have presented a framework for reasoning about analyses and transformations in the presence of RTCG. Our framework consists of an intermediate language (CIR) with explicit support for RTCG, a generalization of control-flow graphs useful for optimizing code templates, and a formal translation from a source language.

This work is only the first step in formalizing compilation for RTCG. We would like to prove the correctness of the translation from Cyclone to CIR and the correctness of analyses and transformations over CIR. Doing so requires a formal semantics for Cyclone and CIR. More speculatively, it would be interesting to extend our framework to languages with first-class code fragments such as 'C. In such languages, templates can be shared by multiple template containers and multiple functions can coordinate to generate a function. We believe these two features would make inter-template optimization analogous to inter-procedural optimization.

We are also working on more substantial benchmarks to better understand the tradeoffs of RTCG in an application setting. Before undertaking an extensive performance evaluation we intend to add key optimizations, such as array-bounds-check elimination, constant folding, copy propagation, dead-code elimination, and instruction scheduling, to those already in the compiler (register allocation, inlining, and null-check elimination). The formal framework presented here has proven crucial for understanding how to fold these optimizations into our compiler.

A Functional translation

We describe a purely functional translation from Mini-Cyclone to CIR. The translation is defined as a function that takes an environment to an environment containing the translated code. The translation language is a λ-calculus with products, records, and lists that is augmented with syntactic sugar to keep the translation palatable. Any standard semantics for the translation language suffices.

This translation looks similar to the one in section 7 (and often uses the same names and syntax), but the meanings of many terms are different because they are functional. This section assumes the reader has read the previous translation and is therefore somewhat more terse.

In the following sections, we describe the environment for the translation, the translation of non-RTCG constructs, and the translation of RTCG constructs.

A.1 Translation environment

The translation of a Mini-Cyclone term is with respect to a global environment, Φ, that contains local environments, $\mathscr{E}$, for each function being translated simultaneously. A global environment is a record of type τ_Φ:

$$\begin{array}{rcl} \tau_\Phi & = & \{ \quad \mathbf{funs} : \text{CIR function list}, \ \mathbf{ids} : \textsc{var} \text{ list}, \\ & & \quad\ \ \mathbf{parents} : \tau_\mathscr{E} \text{ list}, \ \mathbf{active} : \tau_\mathscr{E}, \ \mathbf{children} : \tau_\mathscr{E} \text{ list}\} \\ \tau_\mathscr{E} & = & \{\mathbf{current} : \textsc{label}, \ \mathbf{fun} : \text{block list} \} \end{array}$$

In τ_Φ, the **funs** field contains the set of completed functions and the **ids** field contains the set of "used" variables. The rest of the environment is the same as in Section 7. As in the other translation, we assume the syntax classes for CIR variables, template names, regions names, holes, and instance names are isomorphic. As before, given an element of any class (call it a), we can get the corresponding variable ($\mathscr{V}(a)$), label ($\mathscr{L}(a)$), hole ($\mathscr{H}(a)$), template name ($\mathscr{T}(a)$), code region ($\mathscr{R}(a)$), or instance ($\mathscr{P}(a)$).

Our initial environment, call it Φ_0, is defined as follows:

$$\begin{array}{l} \mathscr{E}_0 = \{\mathbf{current} = l, \mathbf{fun} = \cdot\} \\ \Phi_0 = \{\mathbf{ids} = \text{ids}_0, \mathbf{funs} = \cdot, \mathbf{parents} = \cdot, \mathbf{active} = \mathscr{E}_0, \mathbf{children} = \cdot\} \end{array}$$

For ids_0, we need the list of all x such that x appears in the source program or $x = \mathscr{V}(f)$ where f appears in the source program. We use $\cdot$ for the empty list. $\mathscr{E}_0$ is a placeholder; it is never used.

A.2 Non-RTCG translation

Each top-level function of the input program is translated independently, so to compile a program $P = D_1, \ldots, D_n$, we simply begin with an initial environment, compile each function, and retrieve the value of the **funs** field:

$$[[D_1, \ldots, D_n]] = (([[D_1]]; \cdots ; [[D_n]]) \ \Phi_0).\mathbf{funs}$$

We use dot-notation for record projection and semi-colon for reverse composition ($f ; g = \lambda x.g(f \ x)$).

The translation of a function, $[[D]]$, has type ($\tau_\Phi \to \tau_\Phi$). The resulting environment has a new CIR function that is the translation of D. It also has functions for the `codegen` expressions in D, but the translation of `codegen` is responsible for that. The definition of $[[D]]$ creates an $\mathscr{E}$, uses it to translate the body, and adds the appropriate function to the **funs** field:

$$\begin{array}{l} [[f(x_1, \ldots, x_n) \ s]] \ \Phi = \\ \quad \texttt{let } \mathscr{E} = \{\mathbf{current} = \mathscr{L}(f), \mathbf{fun} = [(\mathscr{L}(f), \mathscr{T}(f), \cdot, \underline{\texttt{jmp}} \circ)]\} \texttt{ in} \\ \quad \texttt{let } \Phi_1 = \Phi[\mathbf{active} = \mathscr{E}] \texttt{ in} \\ \quad \texttt{let } \Phi_2 = [[s]] \ \Phi_1 \texttt{ in} \\ \quad \Phi_2[\mathbf{funs} = \Phi_2.\mathbf{funs} :: \{(x_1, \ldots, x_n)\Phi_2.\mathbf{active}.\mathbf{fun}\}] \end{array}$$

We use the notation $R[\mathbf{label} = F]$ for functional-record update.

https://doi.org/10.1017/S095679680200463X Published online by Cambridge University Press

The translation of statements, $[[s]]$, also has type $(\tau_\Phi \rightarrow \tau_\Phi)$. The translation of expressions, $[[e]]$, has type $(\textsc{var} \rightarrow \tau_\Phi \rightarrow \tau_\Phi)$. The definitions $[[s]]$ and $[[e]]$ are inductive over the structure of Mini-Cyclone statements and expressions. We first present the simpler cases (relying on intuition to get the feeling for the translation), then define the unfamiliar notation, and then present the cases for control flow.

$$\begin{array}{l} [[i]]\ x = \\ \quad \mathit{emit}\ \ x := i \\ \\ [[x]]\ y = \\ \quad \mathit{emit}\ \ y := x \\ \\ [[f]]\ x = \\ \quad \mathit{emit}\ \ x := \mathscr{L}(f) \end{array}$$

$$\begin{array}{l} [[e_1 + e_2]]\ x = \\ \quad \texttt{let}\ x_1 = \mathit{newVar}\ \texttt{then} \\ \quad \texttt{let}\ x_2 = \mathit{newVar}\ \texttt{then} \\ \quad [[e_1]]\ x_1; \\ \quad [[e_2]]\ x_2; \\ \quad \mathit{emit}\ \ x := x_1 + x_2 \end{array}$$

$$\begin{array}{l} [[e_0(e_1, \ldots, e_n)]]\ x = \\ \quad \texttt{let}\ x_0 = \mathit{newVar}\ \texttt{then} \\ \quad \ldots \\ \quad \texttt{let}\ x_n = \mathit{newVar}\ \texttt{then} \\ \quad [[e_0]]\ x_0; \\ \quad \ldots \\ \quad [[e_n]]\ x_n; \\ \quad \mathit{emit}\ \ x := x_0(x_1, \ldots, x_n) \end{array}$$

$$\begin{array}{l} [[x := e]] = \\ \quad [[e]]\ x \end{array}$$

$$\begin{array}{l} [[s_1 ; s_2]] = \\ \quad [[s_1]]; \\ \quad [[s_2]] \end{array}$$

$$\begin{array}{l} [[\texttt{return}\ e]] = \\ \quad \texttt{let}\ x = \mathit{newVar}\ \texttt{then} \\ \quad [[e]]\ x; \\ \quad \mathit{emit}\ \ \underline{\texttt{retn}}\ x \end{array}$$

To make these definitions precise, we first describe the translation of constant integers. The body, $\mathit{emit}\ \ x := i$, appends $x := i$ to the body of the current block. In general, the expression $\mathit{emit}\ \ \iota$ takes an environment, Φ, and produces an identical environment except that the instruction ι is appended to the current block, i.e. the block in $\Phi.\mathbf{active}.\mathbf{fun}$ with label equal to $\Phi.\mathbf{active}.\mathbf{current}$. Similarly, the expression $\mathit{emit}\ \ \chi$ replaces the control transfer of the current block with χ. It is easy to see that with these definitions, $[[i]]\ x$ has the type $\tau_\Phi \rightarrow \tau_\Phi$, as required.

The argument to the emit function, $x\ :=\ i$, is not a proper CIR instruction because x is a metavariable. However, the translation applies $[[e]]$ to an expression that evaluates to a CIR variable, so we let $x\ :=\ i$ mean the CIR syntax obtained by replacing x with the value to which it is bound. We blur this distinction between metavariables and variables for the other CIR constructs analogously.

Now consider $[[e_1 + e_2]]\ x$. The last three lines are already well-defined (recall $f ; g\ =\ \lambda x.g(f\ \ x)$). The rest of the body has two nested occurrences of the form $\texttt{let}\ y = f\ \texttt{then}\ g$. For example, the outer occurence has x_1 for y, newVar for f, and the rest of the body for g. We define $\texttt{let}\ y = f\ \texttt{then}\ g$ to mean $(f\ (\lambda y.\ g))$. That is, f is a function expecting a continuation and $\lambda y.\ g$ is such a continuation.

So newVar has type $(\textsc{var} \rightarrow \tau_\Phi \rightarrow \tau_\Phi) \rightarrow \tau_\Phi \rightarrow \tau_\Phi$; given a continuation and an environment, it returns an environment. newVar calls the continuation with a new CIR variable (not a metavariable) and an environment recording the variable's use:

$$\mathit{newVar}\ k\ \Phi = k\ {''x''}\ \Phi[\mathbf{ids} = \Phi.\mathbf{ids} :: \{''x''\}] \quad (\text{where } ''x'' \in \textsc{var} \setminus \Phi.\mathbf{ids})$$

We have now described all of the notation used above. The remaining non-RTCG constructs manipulate control flow, so they are more complicated:

```
$[[\texttt{if}(e)\ s_1\ \texttt{else}\ s_2]] =$
    let $x = newVar$ then
    $[[e]]\ x$;
    let $(b_0, b_t) = changeBlock$ then
    $[[s_1]]$;
    let $(b_1, b_f) = changeBlock$ then
    $[[s_2]]$;
    let $(b_2, b_m) = changeBlock$ then
    let $d_f = genDest(b_0, b_f)$ then
    let $d_m = genDest(b_1, b_m)$ then
    $emit\ \ b_0 : \underline{\texttt{jnz}}\ x\ ?\circ : d_f$;
    $emit\ \ b_1 : \underline{\texttt{jmp}}\ d_m$;
    $emit\ \ b_2 : \underline{\texttt{jmp}}\ \circ$
```

```
$[[\texttt{while}(e)\ s]] =$
    let $(b_0, b_t) = changeBlock$ then
    let $x = newVar$ then
    $[[e]]\ x$;
    let $(b_1, b_b) = changeBlock$ then
    $[[s]]$;
    let $(b_2, b_e) = changeBlock$ then
    let $d_e = genDest(b_1, b_e)$ then
    let $d_t = genDest(b_2, b_t)$ then
    $emit\ \ b_0 : \underline{\texttt{jmp}}\ \circ$;
    $emit\ \ b_1 : \underline{\texttt{jnz}}\ x\ ?\circ : d_e$;
    $emit\ \ b_2 : \underline{\texttt{jmp}}\ d_t$
```

These cases are quite like their non-functional counterparts except that they use continuations. It remains to define *changeBlock* and *genDest* (in the next section).

```
$changeBlock\ k\ \Phi =$
    let $\mathscr{E} = \Phi.\mathbf{active}$ in
    let $b_{old} = \mathscr{E}.\mathbf{current}$ in
    let $b = newVar$ then
    let $l = \mathscr{L}(b)$ in
    let $t = currentTemplate\ \mathscr{E}$ in
    let $\mathscr{E}' = \{\mathbf{current} = l, \mathbf{fun} = append(\mathscr{E}.\mathbf{fun}, (l, t, \cdot, \underline{\texttt{jmp}}\ \circ))\}$ in
    $k\ (b_{old}, b)\ \Phi[\mathbf{active} = \mathscr{E}']$
```

changeBlock invokes its continuation with a pair of labels. The first is the label of what was the current block. The second is the label of a new empty block added to the end of the active function and made the current block. The new block has the same template name as the old current block. To be precise, *currentTemplate* $\mathscr{E}$ evaluates to the template name of the block in $\mathscr{E}.\mathbf{fun}$ with label $\mathscr{E}.\mathbf{current}$ and *append* puts its second argument at the end of the list of blocks in its first argument.

A.3 RTCG translation

We now explain how the four Mini-Cyclone constructs specific to RTCG are compiled. We begin with `codegen` and use it to explain most of the new concepts:

```
$[[\texttt{codegen}\ f(x_1, \ldots, x_n)s]]\ x\ \Phi_0 =$
    let $\Phi_0 = emit\ \mathscr{R}(f) := \underline{\texttt{start}}\ \mathscr{L}(f)\ \Phi_0$ in
    let $\Phi_0 = emit\ \mathscr{P}(f) := \underline{\texttt{copy}}\ \mathscr{T}(f)\ \underline{\texttt{into}}\ \mathscr{R}(f)\ \Phi_0$ in
    let $\Phi_1 = \Phi_0[\mathbf{parents} = \Phi_0.\mathbf{active} :: \Phi_0.\mathbf{parents}][\mathbf{children} = \cdot]$ in
    let $\Phi_2 = [[f(x_1, \ldots, x_n)s]]\ \Phi_1$ in
    let $\Phi_3 =\ \ \Phi_2\ \ [\mathbf{parents} = tail(\Phi_2.\mathbf{parents})]$
                      $[\mathbf{active} = head(\Phi_2.\mathbf{parents})]$
                      $[\mathbf{children} = \Phi_0.\mathbf{children}]$
    in $(emit\ \ x := \underline{\texttt{end}}\ \mathscr{R}(f))\ \Phi_3$
```

https://doi.org/10.1017/S095679680200463X Published online by Cambridge University Press

The translation creates parent and child code. At the parent level, it allocates a code region, copies the first template, and, after everything else, converts the code region to executable code ($x := \underline{\texttt{end}}\ \mathscr{R}(f)$). At the child level, we translate the function declaration. After translating the child, we use Φ_3 to create an environment for translating the rest of the parent. The **parents** and **active** fields must come from Φ_2 because a `cut` or `fill` in s affects the value of Φ_2.**parents**. The **children** field must come from Φ_0 because the `codegen` may be within a `cut` or `fill`.

The translations for `cut` and `splice` defer the interesting work to auxiliary terms:

$$
\begin{array}{ll}
[[\texttt{cut}\ s]] = & \qquad [[\texttt{splice}\ s]] = \\
\quad \mathit{emit}\ \underline{\texttt{jmp}}\ \circ; & \qquad\quad \uparrow \mathit{newTemplate}; \\
\quad \downarrow [[s]]; & \qquad\quad \uparrow [[s]]; \\
\quad \mathit{newTemplate} & \qquad\quad \uparrow \mathit{emit}\ \underline{\texttt{jmp}}\ \circ
\end{array}
$$

Intuitively, $\downarrow$ gives $[[s]]$ an environment that "shifts" the head of **parents** to **active** and the **active** field to the head of **children** and then "unshifts" after the translation of s. The $\uparrow$ term used in `splice` s "shifts" and "unshifts" in the opposite direction. To be precise, we define:

$$
\begin{array}{rcl}
\mathit{up}\ \Phi & = & \Phi[\mathbf{parents} = \Phi.\mathbf{active} :: \Phi.\mathbf{parents},\ \mathbf{active} = \mathit{head}(\Phi.\mathbf{children}), \\
 & & \ \mathbf{children} = \mathit{tail}(\Phi.\mathbf{children})] \\
\mathit{down}\ \Phi & = & \Phi[\mathbf{children} = \Phi.\mathbf{active} :: \Phi.\mathbf{children},\ \mathbf{active} = \mathit{head}(\Phi.\mathbf{parents}), \\
 & & \ \mathbf{parents} = \mathit{tail}(\Phi.\mathbf{parents})] \\
\downarrow e & = & \mathit{down};\ e;\ \mathit{up} \\
\uparrow e & = & \mathit{up};\ e;\ \mathit{down}
\end{array}
$$

Translation of well-formed source code never applies *head* or *tail* to an empty list.

We use *newTemplate* when moving from parent to child. It creates a new template, puts a new block in it, and emits a copy instruction in the parent.

$$
\begin{array}{l}
\mathit{newTemplate} = \\
\quad \texttt{let}\ f = \mathit{currentFunction}\ \texttt{then} \\
\quad \texttt{let}\ x = \mathit{newVar}\ \texttt{then} \\
\quad \texttt{let}\ (b_0, b_1) = \mathit{changeBlock}\ \texttt{then} \\
\quad \mathit{setTemplateOfCurrent}\ \mathscr{T}(x); \\
\quad \downarrow \mathit{emit}\ \mathscr{P}(x) := \underline{\texttt{copy}}\ \mathscr{T}(x)\ \underline{\texttt{into}}\ \mathscr{R}(f)
\end{array}
$$

Given a template name (t) and an environment, *setTemplateOfCurrent* returns an environment where t is the current block's template name. *currentFunction* retrieves the active function's name (for Φ, the label of Φ.**active**.**fun**).

The translation of `fill` e is straightforward at this point. In the function generating the current one, we translate e and emit a `fill` instruction for a new hole. In the template, we assign the hole to the result.

```
$[[\texttt{fill}\ e]]\ x =$
    $\texttt{let}\ x_1 = newVar\ \texttt{then}$
    $\downarrow [[e]]\ x_1;$
    $\texttt{let}\ x_2 = newVar\ \texttt{then}$
    $\texttt{let}\ t = activeTemplate\ \texttt{then}$
    $\downarrow emit\ \underline{\texttt{fill}}\ \mathscr{P}(t).[\mathscr{H}(x_2)]\ \underline{\texttt{with}}\ x_1;$
    $emit\ \ x := [\mathscr{H}(x_2)]$
```

activeTemplate retrieves the template name of the current block.

Finally, the translation uses $\underline{\texttt{fill}}\ p_1.[h]\ \underline{\texttt{with}}\ p_2.l$ instructions in the correct definition of *genDest* precisely when an inter-template jump occurs:

```
$genDest\ (b_{src}, b_{dst})\ k\ \Phi =$
    $\texttt{let}\ t_{src} = templateOf\ \ \Phi\ b_{src}\ \texttt{in}$
    $\texttt{let}\ t_{dst} = templateOf\ \ \Phi\ b_{dst}\ \texttt{in}$
    $\texttt{if}\ t_{src} = t_{dst}\ \texttt{then}\ k\ b_{dst}\ \Phi$
    $\texttt{else (let}\ h = newVar\ \texttt{then}$
        $\downarrow emit\ \underline{\texttt{fill}}\ \mathscr{P}(t_{src}).[\mathscr{H}(h)]\ \underline{\texttt{with}}\ \mathscr{P}(t_{dst}).b_{dst};$
        $k\ [\mathscr{H}(h)])\ \Phi$
```

templateOf Φ b is the template name of the block in Φ.**active**.**fun** with label b.

Acknowledgments

Many thanks to Anne Rogers and Kathleen Fisher for their careful reading and many helpful suggestions.

References

Aho, A. V., Sethi, R. and Ullman, J. D. (1986) *Compilers, Principles, Techniques and Tools.* Addison-Wesley.

Auslander, J., Philipose, M., Chambers, C., Eggers, S. J. and Bershad, B. N. (1996) Fast, effective dynamic compilation. *ACM Conference on Programming Language Design and Implementation*, pp. 149–159.

Burger, D. (1998) *Hardware techniques to improve the performance of the processor/memory interface.* PhD thesis, University of Wisconsin-Madison.

Burger, D., Austin, T. M. and Bennett, S. (1996) *Evaluating future microprocessors: The simplescalar tool set.* Technical report CS-TR-1996-1308, University of Wisconsin-Madison, Computer Sciences Department.

Consel, C. and Noël, F. (1996) A general approach to run-time specialization and its application to C. *23rd ACM Symposium on Principles of Programming Languages*, pp. 145–156.

Cousineau, G., Curien, P.-L. and Mauny, M. (1985) The categorical abstract machine. In: Jouannaud, J.-P., editor, *Functional Programming Languages and Computer Architecture: Lecture Notes in Computer Science 201*, pp. 50–64. Springer-Verlag.

Davies, R. and Pfenning, F. (1996) A modal analysis of staged computation. *23rd ACM Symposium on Principles of Programming Languages*, pp. 258–270.

https://doi.org/10.1017/S095679680200463X Published online by Cambridge University Press

Engler, D., Hsieh, W. and Kaashoek, M. F. (1996) 'C: A language for fast, efficient, high-level dynamic code generation. *23rd ACM Symposium on Principles of Programming Languages*, pp. 131–144.

Fraser, C. W. and Hanson, D. R. (1995) *A Retargetable C Compiler Design and Implementation.* Benjamin Cummings.

Grant, B., Mock, M., Plilipose, M., Chambers, C. and Eggers, S. J. (1997) *DyC: An expressive annotation-directed dynamic compiler for C.* Technical report UW-CSE-97-03-03, Department of Computer Science and Engineering, University of Washington. (Updated May 12 1999.)

Grant, B., Philipose, M., Mock, M., Chambers, C. and Eggers, S. J. (1999) An evaluation of staged run-time optimizations in DyC. *ACM Conference on Programming Language Design and Implementation*, pp. 293–304.

Hornof, L. and Jim, T. (1999) Certifying compilation and run-time code generation. *ACM Conference on Partial Evaluation and Semantics-based Program Manipulation*, pp. 60–74.

Lee, P. and Leone, M. (1996) Optimizing ML with run-time code generation. *ACM Conference on Programming Language Design and Implementation*, pp. 137–148.

Leone, M. and Lee, P. (1994) Lightweight run-time code generation. *ACM Conference on Partial Evaluation and Semantics-based Program Manipulation*, pp. 97–106.

Lowney, P. G., Freudenberger, S. M., Karzes, T. J., Lichtenstein, W. D., Nix, R. P., O'Donnell, J. S. and Ruttenberg, J. C. (1993) The Multiflow Trace Scheduling compiler. *J. Supercomputing*, **7**(1–2), 51–142.

Morrisett, G., Walker, D., Crary, K. and Glew, N. (1999) From System F to typed assembly language. *ACM Trans. Program. Lang. Syst.* **21**(3), 528–569.

Muchnick, S. S. (1997) *Advanced Compiler Design and Implementation.* Morgan Kaufmann.

Noël, F., Hornof, L., Consel, C. and Lawall, J. L. (1996) *Automatic, template-based run-time specialization: Implementation and experimental study.* Technical report 1065, Institut de Recherche en Informatique et Systémes Aléatoires (IRISA).

Poletto, M., Engler, D. and Kaashoek, M. F. (1997) tcc: A system for fast, flexible and high-level dynamic code generation. *ACM Conference on Programming Language Design and Implementation*, pp. 109–121.

Smith, F. (2002) *Certified run-time code generation.* PhD thesis, Cornell University.

Taha, W. (2000) A sound reduction semantics for untyped CBN multi-stage computation. Or, the theory of MetaML is non-trival (extended abstract). *ACM Conference on Partial Evaluation and Semantics-based Program Manipulation*, pp. 34–43.

Taha, W. and Sheard, T. (1997) Multi-stage programming with explicit annotations. *ACM Conference on Partial Evaluation and Semantics-based Program Manipulation*, pp. 203–217.

Wickline, P., Lee, P. and Pfenning, F. (1998) Run-time code generation and Modal-ML. *ACM Conference on Programming Language Design and Implementation*, pp. 224–235.

https://doi.org/10.1017/S095679680200463X Published online by Cambridge University Press