Compiling for Run-time Code Generation

FREDERICK SMITH, DAN GROSSMAN, GREG MORRISETT
Cornell University
(*e-mail:* {fms,danieljg,jgm}@cs.cornell.edu)

LUKE HORNOF
Transmeta Corporation
(*e-mail:* hornof@transmeta.com)

TREVOR JIM
AT&T Research
(*e-mail:* trevor@research.att.com)

Abstract

Cyclone is a type-safe programming language with explicit support for 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 run-time cost of code generation low, but it requires that optimizations, such as register allocation and code motion, be applied to templates at compile time. This paper describes a principled approach to achieving such optimizations. In particular, we generalize standard flow-graph intermediate representations to support templates, define a formal 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 (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 at compile time. In particular, we can statically verify that any dynamically generated code will be type-correct. Such strong guarantees are particularly important in the RTCG setting where debugging programs and compilers is notoriously difficult.

The original Cyclone compiler used a template-based compilation strategy for RTCG similar to that used by Tempo and DyC. A *template* is a pre-compiled code fragment. At run time, templates are copied together to create new functions. The generating code controls the order and number of the copies, and it may overwrite certain operands in the template, called *holes*, with new values. For example, a jump target will typically be overwritten to reflect the location of the target in the newly generated function.

We were able to achieve good speedups with this template-based approach on a number of small benchmarks. However, our original compiler did not perform optimizations, such as register allocation, loop-invariant removal, copy propagation, or common-subexpression elimination. Rather, it generated fairly naive, stack-based typed assembly language, and as a result, the absolute performance relative to an optimizing compiler was disappointing (see below).

It became clear that to achieve better performance we needed to more aggressively optimize the template code. Furthermore, to keep the overhead of dynamically generating code small, we needed to do as much optimization at compile time as possible. In particular, we needed to perform optimizations such as register allocation *across* template boundaries at compile time, even though we may not know exactly how templates will be stitched together at run time.

Therefore, we set out to adapt traditional flow-graph intermediate representations, dataflow analyses (such as liveness analysis), and optimizations (such as register allocation) to a context with explicit run-time code generation. The goal of this paper is to describe the formal framework we developed and the subtleties we encountered in combining traditional optimizations with RTCG.

Our optimization framework is based on an intermediate language, *Cyclone Intermediate Representation (CIR)*, suitable for performing compile-time inter-template analyses and transformations. The first part of the paper presents a simplified core Cyclone and CIR, and a formal translation between them. Though seemingly straightforward, the translation is subtle: It relies on stacks of contexts, like those found in modal type systems, to track the staging of the input, and on careful orchestration to keep the generating and generated functions synchronized. The second half of the paper explains how we extend dataflow analyses to CIR. We find that many analyses and transformations carry over without modification, but that those involving code motion, such as loop-invariant removal, do not.

Thus, the contributions of this paper include:

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

We are implementing and evaluating a new, optimizing compiler based on our framework. We have implemented a translation from the full Cyclone language to a CIR-based intermediate form, and have implemented a few important analyses and optimizations (notably inlining, null-check elimination, and graph-coloring intra-procedural register allocation). We found the formal framework invaluable for understanding and structuring the code.

Thus far, the results from the new compiler are encouraging: Figure 1 compares the performance of 8 micro-benchmarks and one small application when compiled with our new optimizing compiler versus our old non-optimizing compiler. (See Appendix A for a description of the benchmarks and our experimental methodology.)

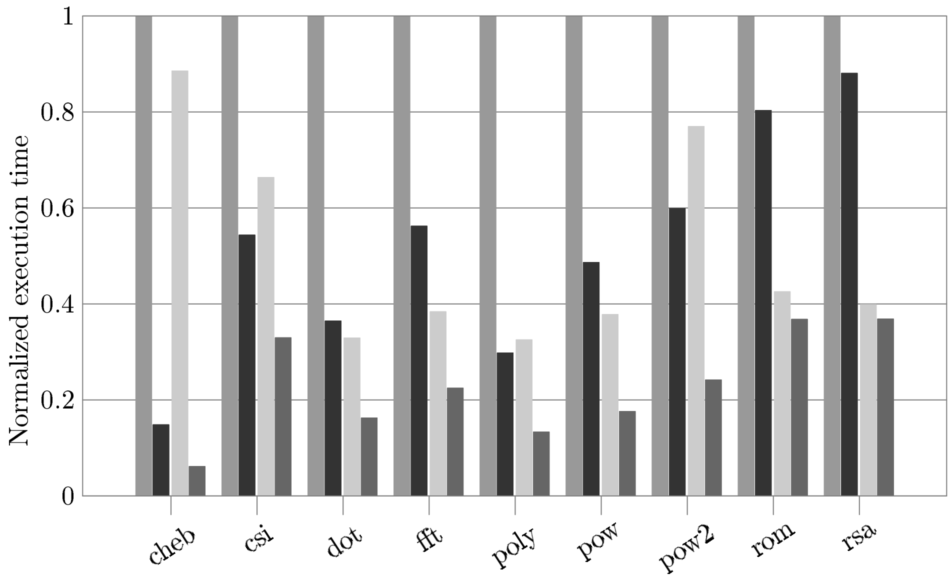

Fig. 1. This plot presents normalized execution times for several benchmark programs. Each program is compiled with our non-optimizing (UNOPT) and optimizing compiler (OPT), each with and without RTCG. From left to right, the four columns represent UNOPT, UNOPT with RTCG, OPT, and OPT with RTCG. The running time for UNOPT serves as the baseline.

The graph demonstrates that (a) a standard, optimizing compiler without RTCG often outperforms a non-optimizing compiler with RTCG, (b) the combination of optimization and RTCG uniformly achieves the best performance. The optimizing Cyclone compiler can be downloaded from `http://www.cs.cornell.edu/talc`.

2 Related Work

The main contribution of this paper is a framework for reasoning about compilation, analyses, and transformations in the presence of RTCG. Prior work has not been directly concerned with these issues although there are a number of implementations supporting RTCG. With regard to compilation strategy, Tempo and DyC are the two most similar to Cyclone.

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 explicitly stage the program. 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 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 an impressive compiler for C that accepts user annotations to direct a binding-time analysis. Like Cyclone, DyC outputs templates. Unlike Cyclone, DyC does not permit dynamically generated code to generate code. In other words, DyC is a two-level language and Cyclone is a multi-level language. DyC also compiles code differently. Instead of compiling an explicitly staged program, DyC tracks an association between user binding-time annotations and variables through optimization. It uses these annotations to stage the program *after* most optimizations have been applied. Some optimizations, such as induction-variable simplification, introduce temporaries that make maintaining this association difficult, so DyC disables these optimizations (Grant *et al.*, 1997).

'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 allows users to choose among different run-time compilers to trade code quality for compilation time. In our terminology, `tcc` delays inter-template optimizations until run time. Although our framework was designed for templates, it does not specify how code is dynamically compiled. Therefore our framework could be adapted for static optimization in `tcc` as well.

Cyclone focuses on giving the user explicit low-level control over RTCG with an operational model close to the actual machine implementation. Systems like Fabius, PML, and MetaML instead provide user control over RTCG in high-level languages. Fabius (Leone & Lee, 1994; Lee & Leone, 1996) pioneered the use of generating extensions. 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 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. Currently the MetaML compiler simply calls itself to generate code dynamically yielding long dynamic compile times but good code quality. Because Fabius, PML, and MetaML do not have an operational model matching the underlying machine, it is harder for the user to predict the costs and benefits of RTCG.

3 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` ($int(dot(int\, v[\,])\ldots)$ indicates RTCG of a function. The generated function will be composed of copies of code fragments (templates) from 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 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 one:

```
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 2, could generate different code when an element of `u` is zero or one.

The RTCG primitives could be implemented using string manipulation and calls

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

to a conventional compiler. Doing so would ensure that the resulting code was well 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 rapidly generate code. As a first step, we translate dynamically generated functions into a collection of templates (code fragments containing holes) in our internal representation. These templates could then be compiled using generating extensions or abstract-syntax trees. Instead, we compile these templates into machine code with holes at compile time.

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 dynamically assembled 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 7). Standard control-flow optimizations can be extended to this representation.

For example, Figure 3 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 3 could not be followed. Such an analysis would allow us to compute more accurate live ranges for the function's variables.

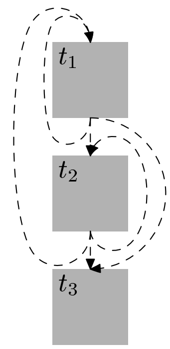

Fig. 3. All ways control might flow between `dot`'s templates. No edges flow out of t_3 because it ends in a `return` statement. The edges (t_2, t_1) and (t_1, t_1) are impossible only because of the way `dot_gen` assembles the templates.

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

Fig. 4. Mini-Cyclone Syntax

In what follows, we make these informal ideas precise by defining a core subset of Cyclone (Section 4), an intermediate representation with templates (Section 5), and a compiler that maps Mini-Cyclone to the intermediate representation (Section 6). We then discuss how analyses and optimizations may be applied to the intermediate representation (Section 7).

4 Mini-Cyclone

Figure 4 presents the abstract syntax for Mini-Cyclone, a subset of Cyclone that serves as this paper's source language. The primitive values are integers (i) 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 statements `cut` s and `splice` s can only occur within a `codegen` expression. The statement `cut` s must occur within a generated function. 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

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

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

$$
\begin{array}{lll}
\text{(instruction)} & \iota ::= & x := v \\
 & \mid & x := v_1 + v_2 \\
 & \mid & x := v(x_1, \ldots, x_n) \\
 & \mid & x := [h] \\
 & \mid & r := \underline{\texttt{start}}\ l \\
 & \mid & p := \underline{\texttt{copy}}\ t\ \underline{\texttt{into}}\ r \\
 & \mid & \underline{\texttt{fill}}\ p.[h]\ \underline{\texttt{with}}\ v \\
 & \mid & \underline{\texttt{fill}}\ p_1.[h]\ \underline{\texttt{with}}\ p_2.l \\
 & \mid & x := \underline{\texttt{end}}\ r \\
\text{(transfer)} & \chi ::= & \underline{\texttt{jmp}}\ d \mid \underline{\texttt{jnz}}\ x\ ?\ d_1 : d_2 \mid \underline{\texttt{retn}}\ x \\
\text{(block)} & B ::= & (l, t, \iota_1 \ldots \iota_n, \chi) \\
\text{(function)} & F ::= & (x_1, \ldots, x_n) B_1, \ldots, B_m \\
\text{(program)} & P ::= & F_1, \ldots, F_n
\end{array}
$$

Fig. 5. CIR Syntax

to append new code to the generated function. The statement `splice` s may only occur 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.

A Mini-Cyclone function takes a fixed number of arguments and executes a body, which is just a statement. For simplicity here, we ignore types and assume that code is well-formed.

5 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 5 shows the syntax for CIR. A CIR program is a collection of functions. A function has a list of parameters and a list of blocks. A block has a label (l), a template name (t), an instruction sequence ($\iota_1, \ldots, \iota_n$), and a control transfer (χ). The first block is the function's entry point; its label serves as the function's name. Assignment ($x := v$), addition ($x := v_1 + v_2$), function call ($x := v(x_1, \ldots, x_n)$), return ($\underline{\texttt{retn}}\ x$), and jump ($\underline{\texttt{jmp}}\ d$) are standard. The conditional transfer ($\underline{\texttt{jnz}}\ x\ ?\ d_1 : d_2$) jumps to d_1 if x is not zero, else it jumps to d_2. The destination $\circ$ is a "fall through" — in block B_i of a function, $\circ$ is equivalent to the label of B_{i+1}.

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.[1] 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 ($x := [h]$).

The instruction r := <u>`start`</u> 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 := <u>`copy`</u> t <u>`into`</u> r puts an instance of t after the instances previously copied into r.[2] The term p becomes a reference to the new instance. The two fill instructions (<u>`fill`</u> $p.[h]$ <u>`with`</u> ...) 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 := <u>`end`</u> 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 <u>`fill`</u> $p_1.[h]$ <u>`with`</u> $p_2.l$ instruction. There is an important exception: If a template's last block has a fall-through destination (such as <u>`jmp`</u> $\circ$), 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 6 shows our `dot_gen` example in CIR (extended with arrays, etc.). The translation of Section 6 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.

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 what the fall-through destinations of 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* reveals that an instance of t_1 would never follow an instance of t_1 or t_2.

[1] 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.

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

```
(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. 6. Translation of the `dot_gen` example (simplified)

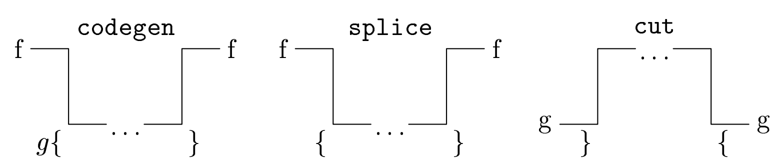

Fig. 7. During translation, these three RTCG constructs change the active function as shown. The higher line denotes the parent, and the lower line the child. The braces denote the start and end of templates.

6 Translation

This section describes a translation from Mini-Cyclone (Section 4) to CIR (Section 5), beginning with some highlights.

Translating a lexically scoped language with RTCG into an unstructured block-based language is challenging because the source language intertwines generated and generating code. In the target language, a function is either executable or a template container, never a mixture. The one-pass top-down translation presented here untangles these levels by maintaining an environment for each level.

Because Cyclone allows generated functions to generate new functions, the translation maintains an ordered sequence of environments. The environment for function f directly precedes that of g exactly when f generates g. In this case, we call f the *parent* of g and g the *child* of f. During the translation, exactly one function is *active*; instructions are emitted into the *active* function. Environments for new functions are added by `codegen`. The RTCG constructs `splice` and `cut` move from parent to child and from child to parent respectively in the existing sequence of environments. Figure 7 shows how the active function changes for each of these constructs. Note that the only difference between `splice` and `codegen` is that `codegen` introduces a new function. The `fill` e construct causes code to be emitted in both the active function and its parent. In the active function, the translation emits a value hole, and in the parent, it emits the code for e followed by a

<u>`fill`</u> instruction with the variable carrying the result of computing e. Mini-Cyclone `fill` instructions correspond to value holes.

By using `cut` statements, the programmer can control which code is emitted and how many times it is emitted. In the `dot` example, the `cut` statement allows the loop to be unrolled. This control implies that the code prior to entering a `cut`, and the code generated inside the `cut` cannot reside in the same template. In general the translation begins a new template on every transition from parent to child (when entering a `splice` or a `codegen`, and when leaving a `cut`).

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

$$\texttt{while}(e)\{\ \texttt{cut}\{\texttt{...}\}\ \}\texttt{...}$$

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-template. If the loop body had not contained a `cut`, then no inter-template 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-template or intra-template 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 issues an inter-template fill ($\underline{\texttt{fill}}\,p_1.[h]\ \underline{\texttt{with}}\,p_2.l$).

The above difficulties make translating a language with RTCG into an unstructured block-based language inherently more complex than standard translations. To explore all the subtleties, we present a full translation from Mini-Cyclone into CIR. In order to emphasize the interesting aspects, some parts are informal; a formal presentation of the translation as a pure monadic-style function over a compilation environment is available (Smith *et al.*, 2000).

6.1 Translation Environment and Preliminaries

The translation of a Mini-Cyclone term uses a global environment, Φ, that contains local environments for each function currently being translated. A global environment is a functional record of type τ_Φ where local environments have type $\tau_\mathcal{E}$.

$$\tau_\Phi \;=\; \{\textbf{parents} : \tau_\mathcal{E}\ \text{list},\ \textbf{active} : \tau_\mathcal{E},\ \textbf{children} : \tau_\mathcal{E}\ \text{list}\}$$

A global environment represents a sequence of local environments, $\mathcal{E}$, of the form:

$$\underbrace{\mathcal{E}_1,\ldots,\mathcal{E}_{k-1}}_{\textbf{parents}},\ \underbrace{\mathcal{E}_k}_{\textbf{active}},\underbrace{\mathcal{E}_{k+1},\ldots,\mathcal{E}_n}_{\textbf{children}}$$

The head of the list **parents** is $\mathcal{E}_{k-1}$ and the head of the list **children** is $\mathcal{E}_{k+1}$. In general, if f is the parent of g, then f immediately precedes g in the sequence. We need a list because `codegen` expressions can nest.

We treat $\tau_\mathcal{E}$ informally because it has the same information as an environment for a conventional compiler: It is a partially generated target function into which the translation (imperatively) adds blocks. $\tau_\mathcal{E}$ also contains a distinguished *current* *block* into which the translation (imperatively) emits instructions. When a function is completely translated, its translation is added to the output by *addFun*.

A tricky aspect of any translation is relating source variables to target variables while easily generating new names. We finesse this issue by asserting:

$$\text{VAR} = \text{CYCVAR} \cup \text{GENVAR} \qquad \text{CYCVAR} \cap \text{GENVAR} = \emptyset$$

Hence variables in GENVAR cannot clash with names in the source program. We use $newVar$ to generate fresh elements of GENVAR.

We assume that the base syntactic classes are isomorphic and that there are functions witnessing these isomorphisms: Given an element of any class (call it a), we can generate the corresponding variable ($\mathcal{V}(a)$), label ($\mathcal{L}(a)$), hole ($\mathcal{H}(a)$), template name ($\mathcal{T}(a)$), code region ($\mathcal{R}(a)$), or instance ($\mathcal{P}(a)$). This technical trick helps 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 use the template name (t) to induce an instance name ($\mathcal{P}(t)$).

6.2 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 all the target functions created.

The translation of a function, $[[D]]$, takes an environment, Φ, as an argument. For top-level functions, Φ should be the empty sequence of function environments. For a D declared in `codegen`, Φ.**parents** will begin with D's parent (see the next section). The role of $[[D]]$ is to translate the function's body under an environment where Φ.**active** is a new function environment. Roughly, *newFun* creates a function environment for $f(x_1, \ldots, x_n)\ s$ in which there is one empty block with label $\mathcal{L}(f)$ and template name $\mathcal{T}(f)$. Using an informal notation:

$$\begin{array}{l} [[f(x_1,\ldots,x_n)\ s]]\ \Phi = \\ \quad \texttt{let}\ \mathcal{E} = \mathit{newFun}\ (f(x_1,\ldots,x_n)\ s)\ \texttt{in} \\ \quad [[s]]\ \Phi[\mathbf{active} = \mathcal{E}]; \\ \quad \mathit{addFun}\ \mathcal{E} \end{array}$$

The main effect of $[[D]]$ is to add a target function that is the translation of D. (As a side-effect, any `codegen` terms in the body of D give rise to target functions too.) Notice we use the notation $\Phi[\mathbf{label} = F]$ for functional record update; the result is a record with the same fields as Φ except field **label** has value F.

The translation of statements, $[[s]]$, also takes a Φ. 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, beginning with the simpler cases.

All of the non-RTCG expression forms appear straightforward to translate. Most of the work lies in generating fresh variables to hold intermediate results. The term *emit* $\Phi\ \iota$ updates Φ such that ι is appended to the end of the active function's current block. Analogously, *emit* Φ χ replaces the control transfer of the current block with χ. Finally, $e_1; e_2$ means, "Do e_1 then e_2."

$$\begin{array}{lll}
[[i]]\ \Phi\ x = & [[e_1 + e_2]]\ \Phi\ x = & [[e_0(e_1, \ldots, e_n)]]\ \Phi\ x = \\
\quad \mathit{emit}\ \Phi\ x := i & \quad \mathtt{let}\ x_1 = \mathit{newVar}\ \mathtt{in} & \quad \mathtt{let}\ x_0 = \mathit{newVar}\ \mathtt{in} \\
 & \quad \mathtt{let}\ x_2 = \mathit{newVar}\ \mathtt{in} & \quad \ldots \\
[[f]]\ \Phi\ x = & \quad [[e_1]]\ \Phi\ x_1; & \quad \mathtt{let}\ x_n = \mathit{newVar}\ \mathtt{in} \\
\quad \mathit{emit}\ \Phi\ x := \mathcal{L}(f) & \quad [[e_2]]\ \Phi\ x_2; & \quad [[e_0]]\ \Phi\ x_0; \\
 & \quad \mathit{emit}\ \Phi\ x := x_1 + x_2 & \quad \ldots \\
[[x]]\ \Phi\ y = & & \quad [[e_n]]\ \Phi\ x_n; \\
\quad \mathit{emit}\ \Phi\ y := x & & \quad \mathit{emit}\ \Phi\ x := x_0(x_1, \ldots, x_n)
\end{array}$$

The following statement cases are just as straightforward:

$$\begin{array}{lll}
[[x := e]]\ \Phi = & [[s_1; s_2]]\ \Phi = & [[\mathtt{return}\ e]]\ \Phi = \\
\quad [[e]]\ \Phi\ x & \quad [[s_1]]\ \Phi; & \quad \mathtt{let}\ x = \mathit{newVar}\ \mathtt{in} \\
 & \quad [[s_2]]\ \Phi & \quad [[e]]\ \Phi\ x; \\
 & & \quad \mathit{emit}\ \Phi\ \underline{\mathtt{retn}}\ x
\end{array}$$

The control-flow constructs (`if` and `while`) are more complicated. Both definitions follow the same strategy: Generate the subterms while remembering the entry and exit blocks for each. Then replace various transfers to create the correct control flow. This order is important for RTCG. As explained earlier, the translation may have to create inter-template control flow at these points. We use the term *genDest* for this purpose. In the absence of RTCG, *genDest* Φ $(b_{src}, b_{dst}) = b_{dst}$; the next section gives the complete definition.

The term *changeBlock* puts a new block in the active function 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 *emit* Φ b: χ replaces the transfer in block b with χ.

$$\begin{array}{ll}
[[\mathtt{if}(e)\ s_1\ \mathtt{else}\ s_2]]\ \Phi = & [[\mathtt{while}(e)\ s]]\ \Phi = \\
\quad \mathtt{let}\ x = \mathit{newVar}\ \mathtt{in} & \quad \mathtt{let}\ (b_0, b_t) = \mathit{changeBlock}\ \Phi\ \mathtt{in} \\
\quad [[e]]\ \Phi\ x; & \quad \mathtt{let}\ x = \mathit{newVar}\ \mathtt{in} \\
\quad \mathtt{let}\ (b_0, b_t) = \mathit{changeBlock}\ \Phi\ \mathtt{in} & \quad [[e]]\ \Phi\ x; \\
\quad [[s_1]]\ \Phi; & \quad \mathtt{let}\ (b_1, b_b) = \mathit{changeBlock}\ \Phi\ \mathtt{in} \\
\quad \mathtt{let}\ (b_1, b_f) = \mathit{changeBlock}\ \Phi\ \mathtt{in} & \quad [[s]]\ \Phi; \\
\quad [[s_2]]\ \Phi; & \quad \mathtt{let}\ (b_2, b_e) = \mathit{changeBlock}\ \Phi\ \mathtt{in} \\
\quad \mathtt{let}\ (b_2, b_m) = \mathit{changeBlock}\ \Phi\ \mathtt{in} & \quad \mathtt{let}\ d_e = \mathit{genDest}\ \Phi\ (b_1, b_e)\ \mathtt{in} \\
\quad \mathtt{let}\ d_f = \mathit{genDest}\ \Phi\ (b_0, b_f)\ \mathtt{in} & \quad \mathtt{let}\ d_t = \mathit{genDest}\ \Phi\ (b_2, b_t)\ \mathtt{in} \\
\quad \mathtt{let}\ d_m = \mathit{genDest}\ \Phi\ (b_1, b_m)\ \mathtt{in} & \quad \mathit{emit}\ \Phi\ \ b_0\colon \underline{\mathtt{jmp}}\ \circ; \\
\quad \mathit{emit}\ \Phi\ \ b_0\colon \underline{\mathtt{jnz}}\ x\ ?\ \circ : d_f; & \quad \mathit{emit}\ \Phi\ \ b_1\colon \underline{\mathtt{jnz}}\ x\ ?\ \circ : d_e; \\
\quad \mathit{emit}\ \Phi\ \ b_1\colon \underline{\mathtt{jmp}}\ d_m; & \quad \mathit{emit}\ \Phi\ \ b_2\colon \underline{\mathtt{jmp}}\ d_t \\
\quad \mathit{emit}\ \Phi\ \ b_2\colon \underline{\mathtt{jmp}}\ \circ &
\end{array}$$

6.3 RTCG Translation

We now explain how the four Mini-Cyclone constructs specific to RTCG are compiled. We already have the machinery needed to translate `codegen`:

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

The translation creates both generating code and generated code. At the generating level, we first allocate the code region ($\mathcal{R}(f) := \underline{\texttt{start}}\ \mathcal{L}(f)$) and copy the child's first template ($\mathcal{P}(f) := \underline{\texttt{copy}}\ \mathcal{T}(f)\ \underline{\texttt{into}}\ \mathcal{R}(f)$). For translating the child, we provide an environment where the current active function is the parent. That way, `cut` statements in s will use the correct local environment. Recall that $[[D]]$ takes care of creating a new function (the child) and considering it active. The translation of s will add code at various levels. After the child has been translated, the parent (that is, the active function of Φ) will not have any more instructions emitted into it that manipulate this child. So the last step is to emit code that converts the code region to executable code ($x := \underline{\texttt{end}}\ \mathcal{R}(f)$). Notice that we use the original Φ except when translating the child function.

For translating the remaining constructs, special notation for shifting the active function will prove useful:

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

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.

The translations for `cut` and `splice` are pleasingly symmetric: (The helper function *newTemplate* is defined below.)

$$\begin{array}{ll}
[[\texttt{cut}\ s]]\ \Phi = & [[\texttt{splice}\ s]]\ \Phi = \\
\quad \mathit{emit}\ \Phi\ \underline{\texttt{jmp}}\ \circ; & \quad \mathit{newTemplate}\ (\mathit{child}(\Phi)); \\
\quad [[s]]\ (\mathit{parent}(\Phi)); & \quad [[s]]\ (\mathit{child}(\Phi)); \\
\quad \mathit{newTemplate}\ \Phi & \quad \mathit{emit}\ (\mathit{child}(\Phi))\ \underline{\texttt{jmp}}\ \circ
\end{array}$$

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 to the child because the parent may choose to copy the code after the transition at different times than the code before the transition. Figure 7 depicts these transitions.

The auxiliary term *newTemplate* creates a new template in the child and a copy instruction in the parent. (*activeFunction* Φ retrieves the active function's name. *setTemplateOfCurrent* Φ t changes the current block's template name to t.)

$$
\begin{array}{l}
newTemplate\ \Phi = \\
\quad \texttt{let}\ f = \mathit{activeFunction}\ \Phi\ \texttt{in} \\
\quad \texttt{let}\ x = newVar\ \texttt{in} \\
\quad \texttt{let}\ (b_0, b_1) = \mathit{changeBlock}\ \Phi\ \texttt{in} \\
\quad \mathit{setTemplateOfCurrent}\ \Phi\ \mathcal{T}(x); \\
\quad \mathit{emit}\ (\mathit{parent}(\Phi))\ \mathcal{P}(x) := \underline{\texttt{copy}}\ \mathcal{T}(x)\ \underline{\texttt{into}}\ \mathcal{R}(f)
\end{array}
$$

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. (*currentTemplate* retrieves the current block's template name.)

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

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 in Section 5, we need such instructions for inter-template jumps. We have just seen that new templates are created when `cut` statements occur inside of other source constructs. This fact is precisely why the control-flow jumps in the translation of `if` and `while` may be inter-template. We now have the machinery to define *genDest* for the general case: We simply check whether the templates of the source and destination blocks are the same (using *templateOf* Φ 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.

$$
\begin{array}{l}
\mathit{genDest}\ \Phi\ (b_{src}, b_{dst}) = \\
\quad \texttt{let}\ t_{src} = \mathit{templateOf}\ \Phi\ b_{src}\ \texttt{in} \\
\quad \texttt{let}\ t_{dst} = \mathit{templateOf}\ \Phi\ b_{dst}\ \texttt{in} \\
\quad \texttt{if}\ t_{src} = t_{dst}\ \texttt{then}\ b_{dst} \\
\quad \texttt{else}\ (\texttt{let}\ x = newVar\ \texttt{in} \\
\qquad \mathit{emit}\ (\mathit{parent}(\Phi))\ \underline{\texttt{fill}}\ \mathcal{P}(t_{src}).[\mathcal{H}(x)]\ \underline{\texttt{with}}\ \mathcal{P}(t_{dst}).b_{dst}; \\
\qquad [\mathcal{H}(x)])
\end{array}
$$

7 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

```
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. 8. 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.

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 container functions because there is no control flow in the containers *per se*. Rather, we are interested in how control may flow in the functions *generated* from the container. We want the CFG for a container function to safely 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.

7.1 Examples

To understand our decision to distinguish intra-template and inter-template edges, consider the two (slightly embellished) Mini-Cyclone functions in Figure 8. The behaviors of f and g are not as important as the similarity of their (simplified) translations shown in Figure 9.

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 10) look quite similar if we ignore the edge

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

Fig. 9. Simplified translations into CIR of the container functions from Figure 8.

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.

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.[3] 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. Second, the translation in Section 6 never fills a hole with a label from the same template.

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.

7.2 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 unconcerned 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).

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

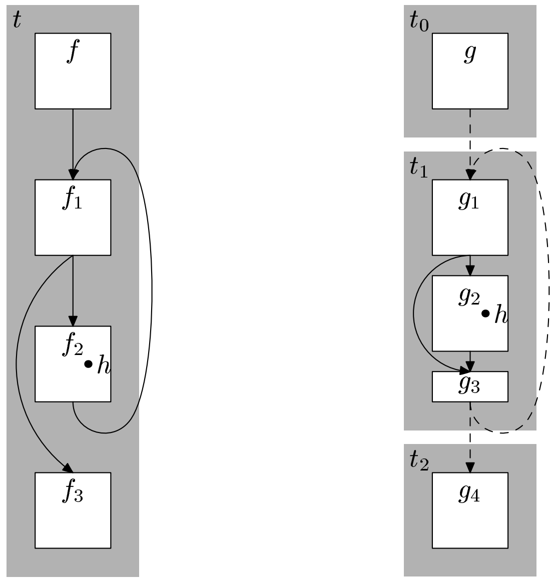

Fig. 10. Control-flow graphs for f and g reflect differences in their generators. Solid arrows denote intra-template edges; dashed arrows denote inter-template edges.

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.[4]

We cannot speak precisely about an analysis being sound without defining a formal semantics for CIR, but we can speak informally: A container function represents a set of functions, specifically, all the 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.

7.3 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

[4] 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.

RTCG, a container function's "interface" to the rest of the program includes how its parents copy templates and fill holes. In other words, any change to the function's templates or holes may affect how a parent needs to generate the function.

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.

7.4 Control-flow Graphs for CIR functions

This section describes the analysis used in the Cyclone compiler to compute generalized CFGs for container functions. Without such an analysis the compiler would have to assume that jump holes might be filled with any label and fall-throughs ending templates might fall through to any template in the function. Because more edges in the CFG result in less precise analyses, we want fewer edges.

There are two distinct kinds of edges in the CFG for F, intra-template edges ($\texttt{intra}(F)$) and inter-template edges ($\texttt{inter}(F)$). The edges in $\texttt{intra}(F)$ represent control flow within a template instance. They arise from label destinations and from intra-template fall-through destinations. The edges in $\texttt{inter}(F)$ represent control flow that may be to another instance.[5] They arise from hole destinations 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 $F = (x_1, \ldots, x_n)B_1, \ldots, B_m$ and $B_i = (l_i, t_i, \ldots, \chi_i)$. Let d be a destination in χ_i. If $d = l_j$ and $t_i = t_j$, then

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

$(l_i, l_j) \in \texttt{intra}(F)$.[6] 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:

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

Instructions add edges by filling holes and copying templates:

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

The first case includes edges introduced by filling holes. The second case is for fall-through destinations: If ι is a copy instruction that copies t, then consider all t' that could have been the most recently copied template when control reaches ι. 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 $\texttt{copypred}(\iota)$ for the set of labels of such "last blocks". Determining $\texttt{copypred}(\iota)$ for each ι 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:

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

For an instruction sequence in $\mathit{parent}(F)$, $\texttt{last}$ denotes the last template copied, if any:

$$\begin{aligned} \texttt{last}(\iota_1, \ldots, \iota_n) &= \begin{cases} \{t\} & \text{if } \iota_n = p := \underline{\texttt{copy}}\ t\ \underline{\texttt{into}}\ r \\ \texttt{last}(\iota_1, \ldots, \iota_{n-1}) & \text{otherwise} \end{cases} \\ \texttt{last}(\cdot) &= \emptyset \end{aligned}$$

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

We extend `last` to blocks via $\texttt{last}(l, t, \iota_1, \ldots, \iota_n, \chi) = \texttt{last}(\iota_1, \ldots, \iota_n)$. We can now state the relevant dataflow equations.

$$\begin{aligned} out(B) &= \begin{cases} \texttt{fallthru}(t) & \text{if } \texttt{last}(B) = \{t\} \\ in(B) & \text{if } \texttt{last}(B) = \emptyset \end{cases} \\ in(B) &= \bigcup_{B' \in \texttt{pred}(B)} out(B') \end{aligned}$$

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 $\iota_{(l,i)}$ denote the i^{th} instruction in block l, we can define `copypred`:

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

Having defined the analysis for the subset of CIR produced by our translation, we now discuss how to generalize the CFG analysis to all of CIR. 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. Note that our translation never produces mutually generating functions.

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.

7.5 Performance of the Analysis

To measure the effect of the analysis, we studied its effectiveness for the benchmarks described in the appendix. Prior to developing our analysis, we were computing a CFG during translation using *ad hoc* techniques, so we used that as a point of comparison. (Separating the translation and CFG creation simplified the compiler.)

For all of the benchmarks, the new analysis includes fewer inter-template edges than the old method. However, the difference is small for most of the programs. The two most substantial changes occurred in `cheb` (from 50 to 25 edges) and `rsa` (from 79 to 52 edges). The overall performance improvement due to the more refined graphs was not substantial except for `cheb`, which sped up by 8.7%. We believe larger more realistic programs will receive greater benefits because they will have more templates and therefore more inter-template edges that can be eliminated.

8 Conclusion and 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.

9 Acknowledgments

We thank Anne Rogers and Kathleen Fisher for their careful reading and many helpful suggestions.

References

Aho, Alfred V., Sethi, Ravi, & Ullman, Jeffrey D. (1986). *Compilers, principles, techniques and tools.* Addison-Wesley.

Auslander, Joel, Philipose, Matthai, Chambers, Craig, Eggers, Susan J., & Bershad, Brian N. 1996 (May). Fast, effective dynamic compilation. *Pages 149–159 of: ACM conference on programming language design and implementation.*

Consel, Charles, & Noël, François. 1996 (Jan.). A general approach to run-time specialization and its application to C. *Pages 145–156 of: 23rd ACM symposium on principles of programming languages.*

Davies, Rowan, & Pfenning, Frank. 1996 (Jan.). A modal analysis of staged computation. *Pages 258–270 of: 23rd ACM symposium on principles of programming languages.*

Engler, Dawson, Hsieh, Wilson, & Kaashoek, M. Frans. 1996 (Jan.). 'C: A language for fast, efficient, high-level dynamic code generation. *Pages 131–144 of: 23rd ACM symposium on principles of programming languages.*

Fraser, Christopher W., & Hanson, David R. (1995). *A retargetable C compiler design and implementation.* Benjamin Cummings.

Grant, Brian, Mock, Markus, Plilipose, Matthai, Chambers, Craig, & Eggers, Susan J. (1997). *DyC: An expressive annotation-directed dynamic compiler for C.* Tech. rept. UW-CSE-97-03-03. Department of Computer Science and Engineering, University of Washington. Updated May 12, 1999.

Grant, Brian, Philipose, Matthai, Mock, Markus, Chambers, Craig, & Eggers, Susan J. 1999 (May). An evaluation of staged run-time optimizations in DyC. *Pages 293–304 of: ACM conference on programming language design and implementation.*

Hornof, Luke, & Jim, Trevor. 1999 (Jan.). Certifying compilation and run-time code generation. *Pages 60–74 of: ACM conference on partial evaluation and semantics-based program manipulation.*

Lee, Peter, & Leone, Mark. 1996 (May). Optimizing ML with run-time code generation. *Pages 137–148 of: ACM conference on programming language design and implementation.*

Leone, Mark, & Lee, Peter. 1994 (June). Lightweight run-time code generation. *Pages 97–106 of: ACM conference on partial evaluation and semantics-based program manipulation.*

Morrisett, Greg, Walker, David, Crary, Karl, & Glew, Neal. (1999). From System F to typed assembly language. *ACM transactions on progamming languages and systems*, **21**(3), 528–569.

Muchnick, Steven S. (1997). *Advanced compiler design and implementation.* Morgan Kaufmann Publishers.

Noël, François, Hornof, Luke, Consel, Charles, & Lawall, Julia L. 1996 (Nov.). *Automatic, template-based run-time specialization: Implementation and experimental study.* Tech. rept. 1065. Institut de Recherche en Informatique et Systémes Aléatoires (IRISA).

Poletto, Massimiliano, Engler, Dawson, & Kaashoek, M. Frans. 1997 (June). tcc: A system for fast, flexible and high-level dynamic code generation. *Pages 109–121 of: ACM conference on programming language design and implementation.*

Rivest, Ronald L., Shamir, Adi, & Adleman, Leonard M. 1978 (Feb.). A method for obtaining digital signatures and public-key cryptosystems. *Pages 120–126 of: Communications of the ACM.*

Smith, Frederick, Grossman, Dan, Morrisett, Greg, Hornof, Luke, & Jim, Trevor. 2000 (Oct.). *Compiling for run-time code generation (extended version).* Tech. rept. Department of Computer Science, Cornell University.

Taha, Walid. 2000 (Jan.). A sound reduction semantics for untyped CBN multi-stage computation. Or, the theory of MetaML is non-trival (extended abstract). *Pages 34–43 of: ACM conference on partial evaluation and semantics-based program manipulation.*

Taha, Walid, & Sheard, Tim. 1997 (June). Multi-stage programming with explicit annotations. *Pages 203–217 of: ACM conference on partial evaluation and semantics-based program manipulation.*

Wickline, Philip, Lee, Peter, & Pfenning, Frank. 1998 (June). Run-time code generation and Modal-ML. *Pages 224–235 of: ACM conference on programming language design and implementation.*

Table A 1. *A brief description of the benchmarks used. The specialized column indicates the value used to dynamically generate code.*

Benchmark	Description	Specialized
cheb	Chebyshev approximation	degree of the polynomial
csi	Cubic spline computation	x-coordinates
dot	Dot product	one input vector, no-zero elements
fft	Fast-fourier transform	size of the input
poly	Compute a polynomial	coefficients
pow	Naive implementation of x^n	n
pow2	Optimized implementation of x^n	n
rom	Romberg integration	interval and precision
rsa	Implementation of RSA encryption	encryption key

Table A 2. *Benchmark execution times, in cycles. The Gen column reports the time to generate the specialized version. The Even column reports the break-even point.*

	Optimizing Compiler				Non-optimizing Compiler			
Benchmark	Plain	RTCG	Gen	Even	Plain	RTCG	Gen	Even
cheb	74309	5170	529384	8	83884	12468	780698	11
csi	25944	12892	20748	2	39075	21251	28874	2
dot	262	130	22882	174	795	290	18211	37
fft	12876	7539	350468	66	33519	18854	542618	38
poly	141	58	4052	49	432	129	7471	25
pow	49	23	802	31	128	62	1718	27
pow2	64	20	811	19	83	50	1894	58
rom	11991	10374	107403	67	28159	22623	169237	31
rsa	17970	16617	263579	195	45028	39668	335211	63

A Experimental Methodology

We chose eight micro-benchmarks and one small application (`rsa` (Rivest *et al.*, 1978)) for this study. A brief description of each benchmark is shown in Table A 1. Of these programs, Tempo used `cheb`, `csi`, `dot`, `fft`, and `rom` for performance measurements (Noël *et al.*, 1996).

All our tests were run on a Pentium II, 300 Mhz with 192 MB of memory. Measurements were taken using the `RDTSC` instruction which provides cycle-level accuracy. Each program was executed ten times and the best execution time was used. The execution times had very little variance, so more runs were not needed.

Figure 1 summarizes the relevant numbers from Table A 2. We have also included the time to generate the specialized code and the number of calls to the generated functions necessary to recover the cost of generation (shown in the Even column).