Compiling for Runtime Code Generation (Extended Version)

Frederick Smith, Dan Grossman, Greg Morrisett, Luke Hornof, and Trevor Jim

Cornell University, {fms,danieljg,jgm}@cs.cornell.edu
Transmeta Corporation, hornof@transmeta.com
AT&T Research, trevor@research.att.com

Abstract. Cyclone is a programming language that provides explicit support for dynamic specialization based on runtime code generation. To generate specialized code quickly, our Cyclone compiler uses a template based strategy in which pre-compiled code fragments are stitched together at runtime. To achieve good performance, the pre-compiled fragments must be optimized. 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 data-flow analysis framework that supports standard optimizations. This extended version contains two mappings to the intermediate representation, a less formal one that emphasizes the novelties of our translation strategy and a purely functional one that is better suited to formal reasoning.

1 Introduction

Runtime code generation (RTCG) is a kind of program specialization that optimizes for short-lived invariants by dynamically generating specialized code. In this setting, where invariants may last only seconds or minutes, compilation time can easily dominate any benefit. Therefore, rapid compilation is critical. *Template-based RTCG* [3, 13, 2] generates code by combining copies of pre-compiled code fragments (templates). Compared to other approaches (see Section 7), template-based RTCG enjoys faster compilation but less optimization.

In earlier work [8], we developed *Cyclone*, a type-safe C-like language with support for explicit RTCG. In that work, we proved the soundness of Cyclone's type system and presented a simple stack-based compiler that translates Cyclone to TAL/T, an extension of Typed Assembly Language [11]. The type-checker for TAL/T statically guarantees that, in addition to standard type-safety properties, no type-unsafe code will be generated at run time. In the context of reliable or secure systems (*e.g.*, operating systems), this guarantee is crucial for assurance.

Though Cyclone allows programmers to dynamically generate specialized code, our previous compiler performed almost no optimization. For instance, we placed all variables on the stack instead of performing register allocation. As a result, the performance of the dynamically generated code was relatively poor

compared to statically generated, but highly-optimized, code. In contrast, the Tempo template-based RTCG system [3, 13] is able to achieve high performance by generating C code with a clever use of labels (to define template boundaries), and then passing the resulting code to GCC. Though an expedient approach, the Tempo designers had to trick GCC into compiling template-based code, and as a result, the compiler does not always generate correct code. Furthermore, the output of GCC cannot be (easily) type-checked. Consequently, Tempo cannot be used in settings where reliability or security are just as important as performance.

We therefore set out to adapt traditional flow-graph intermediate representations, data-flow analyses (such as liveness analysis), and optimizations (such as register allocation) to the template-based RTCG setting, so that we might achieve both high-performance and high-assurance. This project requires a template-aware intermediate representation, a translation from Cyclone to the less structured intermediate form, and a sound and effective framework for performing data-flow-based optimization. We quickly encountered a number of subtleties that suggested these tasks were more difficult than they first appeared.

Our contributions include (1) a formal translation from a source language to an intermediate form, (2) an analysis for computing control-flow graphs for functions containing templates, and (3) a discussion about performing optimizations in the presence of RTCG. Although we describe small theoretical languages, they capture the most interesting aspects of our new optimizing RTCG compiler.

2 Overview

Cyclone is a type-safe C-like language supporting polymorphism, exceptions, and, of course, RTCG. There are four constructs that support RTCG: `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`. The boxes are used to highlight code fragments that are dynamically assembled to produce the specialized function.

```
(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 runtime code generation of a function, `dot`. The `cut` {...} suspends code generation and drops back into running code for `dot_gen`. On each loop iteration, `splice` {...} appends a copy of its contents to the body of `dot`. Furthermore, each copy is specialized using

the `fill` construct to place the values of `u[i]` and `i` in the generated code. Thus, executing `dot_gen({12,24,32})` returns a function like this one:

```
int dot(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 of `dot_gen` could generate different code when an element of `u` is zero, one, or a power of two.

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 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, leveraging the fact that Cyclone does not permit arbitrary operations on code fragments. In particular, we pre-compile code fragments into templates—binary code fragments with "holes" for run-time values—when we initially compile the program. These templates can be quickly copied, and their holes can be filled, to produce a specialized function with low overhead. For example, the compiled `dot_gen` would use three templates: (1) for the function prologue and for initializing `result`, (2) for adding to `result` some constant value (`u[i]`) multiplied by a vector element at some constant offset (`i`), and (3) for returning `result` and the function epilogue.

To achieve good code, we must perform optimizations on the templates, such as register allocation, at compile time. But how can we perform the necessary data-flow analyses, such as live-range analysis, when templates can be dynamically assembled at run-time? The approach we suggest here is to perform data-flow analysis on the *generating* function to construct a *generating graph*—an approximation of the control-flow graph—for the functions that may be generated at run-time. Then we can use the generating graph to analyze the possible data-flow paths of the templates and optimize them.

For example, by analyzing `dot_gen`, we know that, no matter what specialized function it produces, the control flow graph 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. Such an analysis would allow us to compute accurate live ranges for the variables of the function, and to allocate them to registers.

In what follows, we make these informal ideas precise by formally defining a core subset of Cyclone (Section 3), an intermediate representation with explicit support for templates (Section 4), and a compiler that maps Mini-Cyclone to the intermediate representation (Section 5). We then discuss how analyses and optimizations may be applied to the intermediate representation (Section 6).

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

Fig. 1. Mini-Cyclone Syntax

The translation in Section 5 is rather informal; much of the translation environment's structure is unspecified and target code is "emitted" in an undefined imperative fashion. In the appendix we present a formal translation, independent of the first though identical in its overall strategy, which completely describes translation environments. The translation itself is a pure monadic-style function over such environments. Such formality should facilitate formal reasoning about properties of the translation (for example, correctness), but we leave such reasoning to future work.

3 Mini-Cyclone

Figure 1 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 uses the evaluation of e as a constant in the enclosing code fragment.

The standard statements (assignment, composition, conditional, loop, and return) all have their usual meaning. `cut` s must occur within a code fragment for a generated function. Its meaning is to execute s in the generating function while the generated function is being generated. `splice` s must occur within a `cut`. It treats s as a code fragment belonging to the function that was being generated when the enclosing `cut` began executing. A copy of s is placed after the code that has already been generated.

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.

4 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.

$$
\begin{array}{rl}
i & \in \mathcal{Z} \\
x & \in \textsc{var} \\
l & \in \textsc{label} \\
h & \in \textsc{hole} \\
t & \in \textsc{template} \\
r & \in \textsc{region} \\
p & \in \textsc{instance}
\end{array}
$$

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

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

Fig. 2. CIR Syntax

Figure 2 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 (template) container. 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 $r := \underline{\texttt{start}}\ l$ allocates a code region and binds it to r. All templates copied into r should come from the (container) function named l.

[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.

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

(v) (dot, t1, result := 0,
                 jmp ∘)
    (l1, t2,     x := [h1]
                 i := [h2]
                 tmp := x * v[i]
                 result := result + tmp,
                 jmp ∘)
    (l2, t3,     , retn result)
```

Fig. 3. Translation of the `dot_gen` example (simplified)

The instruction $p :=$ `copy` t `into` r puts an instance of t after the instances previously copied into r.[2] p becomes a reference to the new instance. The two fill instructions (`fill` $p.[h]$ `with` ...) 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 put inter-template jumps in the generated code. Finally, $x :=$ `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 `fill` $p_1.[h]$ `with` $p_2.l$ instruction. There is an important exception: If a template's last block has a fall-through destination (such as `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 3 shows our `dot_gen` example in CIR (extended with arrays, etc.). The translation of Section 5 would introduce more variables but would otherwise produce similar output. Notice the function *dot* contains holes, 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 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 *dot*'s epilogue, 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

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

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.

5 Translation

This section describes a translation from Mini-Cyclone (Section 3) to CIR (Section 4). We begin 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 we present untangles the levels by maintaining an environment for each level.

If a function f contains "`codegen` g ...," then we call f the *parent* of g and g a *child* of f. During the translation, exactly one function is *active*; instructions are emitted into the active function. Entering a `cut` activates the active function's parent, and entering a `splice` activates the active function's child. Entering a `codegen` is like entering a `splice` except that it spawns a new child and then activates the child. For `fill` e, we emit a value hole and then activate the parent just long enough to translate e and emit a `fill` instruction. Notice that an explicit Mini-Cyclone `fill` always corresponds to a value hole.

All the code in a template must be copied into a code region at once. Consequently, whenever two pieces of code might not be copied consecutively, we must introduce a new template. Our translation conservatively creates new templates on each transition from parent to child (when entering a `splice` or `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 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-template. If the loop body had not contained a `cut`, then no inter-template jump would have been necessary. After translating the loop body, the translation checks if the template following the loop is the same as the template for the guard. If not, then 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 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 more formal translation is given in the appendix.

5.1 Translation Environment and Preliminaries

The translation of a Mini-Cyclone term is with respect to a global environment, Φ, that contains local environments, $\mathcal{E}$, for each function being simultaneously

translated. A global environment is a (functional) record of type τ_Φ where a local environment has 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 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 we (imperatively) add blocks. It also contains a distinguished *current block* into which we (imperatively) emit instructions. When we are done generating a function, we use *addFun* $\mathcal{E}$ to add it to the output.

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 the function $newVar$ to generate fresh elements of GENVAR.

We assume that the base syntactic classes are isomorphic and that we have functions witnessing these isomorphisms: Given an element of any class (call it a), we can get 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 only one instance of a template is live at a time, so we use the template name (t) to induce an instance name ($\mathcal{P}(t)$).

5.2 Non-RTCG Translation

Each top-level function of the input program is translated independently, so to compile a program, we simply compile each source function and collect 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:

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

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* $\Phi\ \chi$ replaces the control transfer of the current block with χ. Finally, $e_1; e_2$ means, "Do e_1 then e_2."

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

The following statement cases are just as straightforward:

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

The control-flow constructs (`if` and `while`) are more complicated. Both definitions follow the same strategy: Generate the subterms while remembering the entry 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, $\mathit{genDest}(b_{src}, b_{dst}) = b_{dst}$; the next section gives a 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* $\Phi\ \ b\colon \chi$ replaces the transfer in block b with χ.

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

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

5.3 RTCG Translation

We now explain how the four Mini-Cyclone constructs specific to RTCG are compiled. We already have all the machinery we need 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:

$$
\begin{array}{l}
\uparrow \Phi = \{\mathbf{parents} = \Phi.\mathbf{active} :: \Phi.\mathbf{parents},\quad \mathbf{active} = \mathit{head}(\Phi.\mathbf{children}), \\
\qquad\quad \mathbf{children} = \mathit{tail}(\Phi.\mathbf{children})\} \\
\downarrow \Phi = \{\mathbf{children} = \Phi.\mathbf{active} :: \Phi.\mathbf{children}, \mathbf{active} = \mathit{head}(\Phi.\mathbf{parents}), \\
\qquad\quad \mathbf{parents} = \mathit{tail}(\Phi.\mathbf{parents})\}
\end{array}
$$

So $\uparrow \Phi$ makes the child active and $\downarrow \Phi$ makes the parent active. Translation of well-formed source code will never apply *tail* to an empty list.

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

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

Recall that the purpose of `cut` s is to execute s in the parent of the function in which the `cut` appears. Dually, `splice` s puts the code fragment s in the child of the function in which the `splice` appears. 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 put a new template in the child 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. (*activeFunction* Φ retrieves the active function's name. *setTemplateOfCurrent* $\Phi\ t$ changes the current block's template name to t.)

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

The translation of `fill` e is straightforward at this point. In the parent, we translate e and emit a `fill` instruction for a new hole. In the child, we emit a value hole. (*currentTemplate* retrieves the current block's template name.)

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

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 4, 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 easily *genDest* for the general case: 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.

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

6 Optimization

A goal of our research is to study the interaction between RTCG and optimization. By careful design of CIR, we have made the application of traditional data-flow optimizations straightforward. These optimizations can be split into an analysis and a transformation phase. We consider these phases individually.

6.1 Analysis

A preliminary step in data-flow analyses is to build a control-flow graph (CFG) that conservatively approximates the dynamic control flow of the function. For a conventional language, we just put an edge between a block and the blocks that its control transfer mentions. In the presence of RTCG with holes or fall-throughs for destinations, control transfers are no longer apparent. In this section, we describe how to compute an appropriate CFG for any CIR function.

A CFG for a function F, which we write CFG(F), is a directed graph where the nodes are the labels in F. To compute CFG(F), we start with an empty graph and add edges. Adding edges for intra-template control transfers is conventional. For inter-template control transfers, we need to approximate how generating code might create control flow among F's templates. Such control flow can result from filling holes in transfer instructions. It can also result from copying templates that end with fall-through destinations. One conservative approach would be to assume that a hole destination could be filled with any label of any instance of F and that F's templates could be copied in any order.

By analyzing the parent of F (that is, the function containing $r := \underline{\texttt{start}}\ f$ where f is the name of F), we can compute a less conservative CFG. Our translation has the property that executable functions have no parents and non-executable functions have exactly one parent. We denote F's parent by $F\downarrow$. We explain how to extend the analysis to multiple parents at the end of the section.

We begin with the intra-template edges. Let $F = (x_1, \ldots, x_n)B_1, \ldots, B_m$ and $B_i = (l_i, t_i, \iota_{i1} \ldots \iota_{ip}, \chi_i)$. The intra-template destinations for block B_i are:

$$\begin{aligned}
\texttt{intra}_i([h]) &= \emptyset \\
\texttt{intra}_i(l_j) &= \begin{cases} \{l_j\} & \text{if } t_i = t_j \\ \text{undefined} & \text{otherwise} \end{cases} \\
\texttt{intra}_i(\circ) &= \begin{cases} \{l_{i+1}\} & \text{if } i < m \text{ and } t_i = t_{i+1} \\ \emptyset & \text{otherwise} \end{cases}
\end{aligned}$$

Hole destinations are handled by the inter-template part of the algorithm (even though a parent may fill the hole with a label in the same instance). An explicit destination must be to a block in the same template, a fact the undefined case emphasizes. A fall-through destination is an intra-template transfer if and only if the next block in the function belongs to the same template. We extend the definition to transfers ($\texttt{intra}_i(\chi)$) and functions ($\texttt{intra}(F)$) by taking the union of $\texttt{intra}_i(d)$ over all constituent destinations, d.

For functions without parents (executable functions), $\text{CFG}(F) = \texttt{intra}(F)$. For those with parents (container functions), $\text{CFG}(F) = \texttt{intra}(F) \cup \texttt{inter}(F)$. So we now turn to $\texttt{inter}(F)$, assuming we have $\text{CFG}(F\downarrow)$. That is, we compute the parent's CFG and use it to compute the child's CFG. For now, assume that $F\downarrow$ contains exactly one $r := \underline{\texttt{start}}\ l$ instruction. Our goal is to determine what edges we must add to $\text{CFG}(F)$ because of the way $F\downarrow$ manipulates the templates. We assume every instruction in $F\downarrow$ is reachable, so the edges added are the union of the edges each instruction adds:

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

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 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 the labels of such "last blocks". Determining $\texttt{copypred}(\iota)$ for each ι in $F\downarrow$ is a data-flow problem. We define the function $\texttt{fallthru}(t)$ for a template in function 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, `last` denotes the last template copied, if any:

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

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 data-flow equations.

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

These equations are solvable using standard techniques [1, 12] with $in(B) = \emptyset$ for the first block of $F \downarrow$. Note that the definition of $in(B)$ is where we assume we have already determined CFG($F \downarrow$). Using the data-flow solution and letting $\iota_{(l,i)}$ denote the ith instruction in block l, we can define `copypred`:

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

We now discuss how to generalize the CFG analysis. 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 one — because code regions and instance pointers are second-class constructs, we always know what code region an instruction affects. The interesting extension is for when functions are mutually generating. That is, nothing in CIR prevents a cycle in the graph in which the 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.

6.2 Transformation

Although analyses are little affected by RTCG, transformations require care. For example, dead-code elimination might eliminate code containing a hole. Doing so requires removing the attendant fills in the parent. Similarly, if a hole is duplicated, the fills must be duplicated. Our compiler summarizes the relevant transformation effects and then uses this summary to update the parent.

Unfortunately, treating holes as constants can be too aggressive: Each time a template is copied, its holes may be filled with *different* values. For instance, the block l_1 is in a loop in CFG(*dot*) (see Figure 3) and (if holes are constant) i is invariant. However, hoisting the instruction $i := [h_2]$ out of the loop would not preserve the program's semantics. The hoist would have been valid had the source and destination of the transfer been in the same instance (equivalently, if the edge is in `intra`(*dot*)). A principled understanding of transformations will

require having two kinds of edges in CFG – those from `intra` and those from `inter`. For now, our compiler does not move holes across templates.

Note that a sound transformation of a parent will not require updating a child: Because the parent treats the child as data, any correct optimization will copy the templates in the same order. However, optimizing a parent may reveal a less conservative CFG for a child, thus enabling child optimizations.

7 Related Work

Compared to other RTCG systems, Cyclone has a unique combination of features: type-safety, a simple operational model, and user-level control. 'C [5, 14] is an unsafe language providing rich low-level control over RTCG. Users can choose among different run-time compilers to trade code quality for compilation time. In essence, more costly compilers delay inter-template optimizations until run time. MetaML [16, 15] and PML [17, 4] provide type safety and user-level control but do not have an operational model matching the underlying machine. Therefore, the costs and benefits of RTCG are harder to predict. Fabius [10, 9], Tempo [13, 3], and DyC [2, 7, 6] all apply RTCG automatically based on user annotations, thus relieving the user of low-level control.

Our contribution is a formal description of compilation and optimization for template-based RTCG. Prior work has used ad hoc techniques. Tempo produces template C code that is compiled to object code by GCC. From the resulting object code, the Tempo run-time system extracts templates. This approach is easier to implement, but is less robust, than writing a full compiler. DyC is an impressive optimizing compiler with pervasive support for RTCG. Most data-flow optimizations are performed *before* the program is staged, so it is necessary only to preserve the user annotations during optimization. This preservation is sometimes difficult, so some optimizations are disabled [6].

8 Conclusion

In this paper, we have presented a formal translation from a language (Mini-Cyclone) with explicit RTCG to an intermediate language (CIR), and we have shown how to apply standard data-flow optimizations. Hopefully, formalizing these techniques will allow other compilers to integrate RTCG more easily.

We have implemented an optimizing compiler for a realistic source language with RTCG; this paper distills the language and compiler down to its essence. Our system currently performs only a small set of optimizations, notably register allocation and null-check elimination. Nonetheless, initial results are promising: For an implementation of RSA, we applied RTCG to code accounting for 30% of the execution time and obtained a 10% speedup over the unspecialized program.

References

1. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques and Tools.* Addison-Wesley, 1986.

2. Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad. Fast, effective dynamic compilation. In *ACM Conference on Programming Language Design and Implementation*, pages 149–159, May 1996.
3. Charles Consel and François Noël. A general approach to run-time specialization and its application to C. In *Twenty-Third ACM Symposium on Principles of Programming Languages*, pages 145–156, January 1996.
4. Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In *Twenty-Third ACM Symposium on Principles of Programming Languages*, pages 258–270, January 1996.
5. Dawson Engler, Wilson Hsieh, and M. Frans Kaashoek. 'C: A language for fast, efficient, high-level dynamic code generation. In *Twenty-Third ACM Symposium on Principles of Programming Languages*, pages 131–144, January 1996.
6. Brian Grant, Markus Mock, Matthai Plilipose, Craig Chambers, and Susan J. Eggers. 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.
7. Brian Grant, Matthai Philipose, Markus Mock, Craig Chambers, and Susan J. Eggers. An evaluation of staged run-time optimizations in DyC. In *ACM Conference on Programming Language Design and Implementation*, pages 293–304, May 1999.
8. Luke Hornof and Trevor Jim. Certifying compilation and run-time code generation. In *ACM Conference on Partial Evaluation and Semantics-Based Program Manipulation*, pages 60–74, January 1999.
9. Peter Lee and Mark Leone. Optimizing ML with run-time code generation. In *ACM Conference on Programming Language Design and Implementation*, pages 137–148, May 1996.
10. Mark Leone and Peter Lee. Lightweight run-time code generation. In *ACM Conference on Partial Evaluation and Semantics-Based Program Manipulation*, pages 97–106, June 1994.
11. Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Progamming Languages and Systems*, 21(3):528–569, May 1999.
12. Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.
13. François Noël, Luke Hornof, Charles Consel, and Julia L. Lawall. Automatic, template-based run-time specialization: Implementation and experimental study. Technical Report 1065, Institut de Recherche en Informatique et Systémes Aléatoires (IRISA), November 1996.
14. Massimiliano Poletto, Dawson Engler, and M. Frans Kaashoek. tcc: A system for fast, flexible and high-level dynamic code generation. In *ACM Conference on Programming Language Design and Implementation*, pages 109–121, June 1997.
15. Walid Taha. A sound reduction semantics for untyped CBN mutli-stage computation. Or, the theory of MetaML is non-trival (extended abstract). In *ACM Conference on Partial Evaluation and Semantics-Based Program Manipulation*, pages 34–43, January 2000.
16. Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *ACM Conference on Partial Evaluation and Semantics-Based Program Manipulation*, pages 203–217, June 1997.
17. Philip Wickline, Peter Lee, and Frank Pfenning. Run-time code generation and Modal-ML. In *ACM Conference on Programming Language Design and Implementation*, pages 224–235, June 1998.

A Formal Translation

We describe a formal translation from Mini-Cyclone (Section 3) to CIR (Section 4). The translation is defined as a function that takes an empty environment to an environment containing the translated code. The translation language is the λ-calculus with sums, products, records, and lists; augmented with syntactic sugar to keep the translation palatable. We omit a semantics for the translation language; any standard one suffices. The translation does not rely on Section 5; some common material is repeated so that this formal translation is self-contained.

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, $\mathcal{E}$, for each function being simultaneously translated. There may be more than one because of RTCG. A global environment is a record of type τ_Φ and a local environment is a record of type $\tau_\mathcal{E}$.

$$\begin{aligned}\tau_\Phi = \{\ &\textbf{funs} : \text{CIR function set},\ \textbf{ids} : \textsc{genvar}\text{ set},\\ &\textbf{parents} : \tau_\mathcal{E}\text{ list},\ \textbf{active} : \tau_\mathcal{E},\ \textbf{children} : \tau_\mathcal{E}\text{ list}\}\\ \tau_\mathcal{E} = \{&\textbf{current} : \textsc{label},\ \textbf{blocks} : \text{block list }\}\end{aligned}$$

In τ_Φ, the **funs** field contains the set of completed functions and the **ids** field contains the set of generated variables. These fields begin empty and grow as the translation proceeds. The **parents**, **active**, and **children** fields contain the environments for partially compiled functions. The **active** field contains the information for the function currently being compiled. These fields represent a sequence 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 (that is, f contains a `codegen` g expression), then f immediately precedes g in the sequence. We need a list because `codegen` expressions can nest.

In $\tau_\mathcal{E}$, the **current** field identifies the block into which instructions are currently being emitted. The **blocks** field contains the list of blocks that will, when completed, be the function's body. We preserve two invariants for local environments: the labels of the blocks in **blocks** are distinct, and **blocks** contains a block with the label in **current**. From a local environment, we can recover the name of the function (it is the label of the first block in **blocks**) and the name of the current template (it is the current block's template).

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 fresh variables in GENVAR cannot clash with names in the source program. We assume that the base syntactic classes are isomorphic to each other:

$$\text{VAR} \cong \text{LABEL} \cong \text{HOLE} \cong \text{TEMPLATE} \cong \text{REGION} \cong \text{INSTANCE}$$

We further assume that we have functions witnessing these isomorphisms: Given an element of any class (call it a), we can get 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 ($\mathcal{P}(a)$). Our translation exploits these isomorphisms in subtle ways. For example, the translation has the property that only one instance of a template is live at a time, so we use the template name (t) to induce an instance name ($\mathcal{P}(t)$).

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 finally retrieve the value of the **funs** field:

$$\begin{array}{l}
[[D_1, \ldots, D_n]] = \\
\quad \texttt{let}\ \mathcal{E}_0 = \{\textbf{current} = l, \textbf{blocks} = \cdot\}\ \texttt{in} \\
\quad \texttt{let}\ \Phi_0 = \{\textbf{ids} = \emptyset, \textbf{funs} = \emptyset, \textbf{parents} = \cdot, \textbf{active} = \mathcal{E}_0, \textbf{children} = \cdot\}\ \texttt{in} \\
\quad (([[D_1]]; \cdots ; [[D_n]])\ \Phi_0).\textbf{funs}
\end{array}$$

We use dot-notation for record projection, semi-colon for reverse composition ($f; g = \lambda x.g(f\ x)$), and $\cdot$ for the empty list. $\mathcal{E}_0$ is a placeholder; it is never used.

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` (see the next section) is responsible for that. The definition of $[[D]]$ creates an $\mathcal{E}$, uses it to translate the body, and adds the appropriate function to the **funs** field:

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

We use the notation $R[\textbf{label} = F]$ for functional-record update; the result is a record with the same field values as R except field **label** has value F.

The translation of statements, $[[s]]$, also has type $(\tau_\Phi \to \tau_\Phi)$. The translation of expressions, $[[e]]$, leaves the result in a CIR variable provided as input, so its type is $(\text{VAR} \to \tau_\Phi \to \tau_\Phi)$. 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. We begin with the simpler cases, relying on intuition to get the feeling for the translation. Then we define the unfamiliar notation. Finally, we present the more difficult cases.

All of the non-RTCG expression forms appear straightforward to translate. The main responsibility is to generate fresh variables to hold intermediate results:

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

The following statement cases are also relatively straightforward:

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

To make the meaning of 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[3] of the active function. Analogously, the expression $\mathit{emit}\ \chi$ replaces the control transfer of the current block with χ.

Notice that the argument to the emit function, $x := i$, is not a proper CIR instruction because x is a meta-variable. However, the translation will apply $[[e]]$ to (an expression that evaluates to) a CIR variable, so we define $x := i$ to mean the CIR syntax obtained by replacing x with the value to which it is bound. We blur this distinction between meta-variables 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.

Hence newVar has type $(\textsc{var} \to \tau_\Phi \to \tau_\Phi) \to \tau_\Phi \to \tau_\Phi$; given a continuation and an environment, it returns an environment. newVar invokes the continuation with a new variable and an environment that remembers the used variable:[4]

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

[3] That is, the block in $\Phi.\mathbf{active.blocks}$ with label equal to $\Phi.\mathbf{active.current}$.

[4] We write $''x''$ to emphasize that it is a CIR variable, not a meta-variable.

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

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

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

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. For example, the translation of conditionals uses b_0 to hold the label of the block in which the translation of e ends and b_t to hold the label of the block in which the s_1 begins. The terms of the form $emit\ b\colon \chi$ are defined to mean that the environment they produce is like the one they consume (Φ) except that the block in $\Phi.\mathbf{active}.\mathbf{blocks}$ with label b now has transfer χ.

In the absence of RTCG, we can let $genDest\ (b_{src}, b_{dst})\ k\ \Phi = k\ b_{dst}\ \Phi$. The next section explains why this definition is insufficient and replaces it. *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.

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

$currentTemplate\ \mathcal{E}$ evaluates to the template name of the block in $\mathcal{E}.\mathbf{blocks}$ with label $\mathcal{E}.\mathbf{current}$. *append* appends two lists.

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:

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

The translation creates both generating code and generated code. At the generating level, it allocates the code region ($\mathcal{R}(f) := \underline{\texttt{start}}\ \mathcal{L}(f)$); copies the first template ($\mathcal{P}(f) := \underline{\texttt{copy}}\ \mathcal{T}(f)\ \underline{\texttt{into}}\ \mathcal{R}(f)$); and, after everything else, converts the code region to executable code ($x := \underline{\texttt{end}}\ \mathcal{R}(f)$). At the generated level, we translate the function declaration. This translation needs the generating code's environment in order to translate `cut` statements. Therefore, we give it an environment where the first element of **parents** is the active function's environment.

After translating the generated function, we create an environment for translating the rest of the parent. The **parents** and **active** fields of Φ_3 must be set from Φ_2 rather than Φ_0 because any top-level `cut` or `fill` constructs in s affect the value of $\Phi_2.\mathbf{parents}$. The **children** field must come from Φ_0 because the `codegen` may be within a `cut` or `fill`.

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

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

Recall that the purpose of `cut` s is to execute s in the parent of the function in which the `cut` appears. So $\downarrow$ 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:

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

Translation of well-formed source code will never apply *tail* to an empty list. Note that Section 5 defines $\downarrow$ and $\uparrow$ quite differently.

With this notation, we can explain the translation of `cut` and `splice`. When moving from child to parent, we emit a $\underline{\texttt{jmp}}\ \circ$ to end the current block. The

newTemplate function performs the actions required 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. We start a new template because control flow in the parent may cause the generated child code to be copied 0, 1, or multiple times. We emit the copy instruction before translating the template's body so that it will precede fills emitted while translating the body.

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

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 $head(\Phi.\textbf{active}.\textbf{blocks})$).

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.

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

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:

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

templateOf $\Phi\ b$ is the template of the block in $\Phi.\textbf{active}.\textbf{blocks}$ with label b.

We have now presented the entire translation. A correctness proof is beyond the scope of this work, but we sketch what an argument would look like. First, it remains to give a formal semantics to Mini-Cyclone and CIR; the semantics for the former would be similar to previously published work [8]. We could then extend the translation to work over program states (as opposed to programs). We would then hope to establish a bisimulation relation.