Certifying Compilation and Run-time Code Generation

Luke Hornof Trevor Jim

Computer and Information Science Department
University of Pennsylvania
Philadelphia, PA 19103
{hornof,tjim}@cis.upenn.edu

Abstract

A certifying compiler takes a source language program and produces object code, as well as a "certificate" that can be used to verify that the object code satisfies desirable properties, such as type safety and memory safety. Certifying compilation helps to increase both compiler robustness and program safety. Compiler robustness is improved since some compiler errors can be caught by checking the object code against the certificate immediately after compilation. Program safety is improved because the object code and certificate alone are sufficient to establish safety: even if the object code and certificate are produced on an unknown machine by an unknown compiler and sent over an untrusted network, safe execution is guaranteed as long as the code and certificate pass the verifier.

Existing work in certifying compilation has addressed statically generated code. In this paper, we extend this to code generated at run time. Our goal is to combine certifying compilation with run-time code generation to produce programs that are both verifiably safe and extremely fast. To achieve this goal, we present two new languages with explicit run-time code generation constructs: Cyclone, a type safe dialect of C, and TAL/T, a type safe assembly language. We have designed and implemented a system that translates a safe C program into Cyclone, which is then compiled to TAL/T, and finally assembled into executable object code. This paper focuses on our overall approach and the front end of our system; details about TAL/T will appear in a subsequent paper.

1 Introduction

1.1 Run-time specialization

Specialization is a program transformation that optimizes a program with respect to invariants. This technique has been shown to give dramatic speedups on a wide range of applications, including aircraft crew planning programs, image shaders, and operating systems [4, 11, 17]. *Run-time* specialization exploits invariants that become available during the execution of a program, generating optimized code on the fly. Opportunities for run-time specialization occur when dynamically changing values remain invariant for a period of time. For example, networking software can be specialized to a particular TCP connection or multicast tree.

Run-time code generation is tricky. It is hard to correctly write and reason about code that generates code; it is not obvious how to optimize or debug a program that has yet to be generated. Early examples of run-time code generation include self-modifying code, and ad hoc code generators written by hand with a specific function in mind. These approaches proved complicated and error prone [14].

More recent work has applied advanced programming language techniques to the problem. New source languages have been designed to facilitate run-time code generation by providing the programmer with high-level constructs and having the compiler implement the low-level details [15, 21, 22]. Program transformations based on static analyses are now capable of automatically translating a normal program into a run-time code generating program [6, 10, 12]. And type systems can check run-time code generating programs at compile time, ensuring that certain bugs will not occur at run time (provided the compiler is correct) [22, 25].

These techniques make it easier for programmers to use run-time code generation, but they do not address the concerns of the compiler writer or end user. The compiler writer still needs to implement a correct compiler—not easy even for a language without run-time code generation. The end user would like some assurance that executables will not crash their machine, even if the programs generate code and jump to it—behavior that usually provokes suspicion in security-concious users. We will address both of these concerns through another programming language technique, *certifying compilation*.

1.2 Certifying compilation

A certifying compiler takes a source language program and produces object code and a "certificate" that may help to show that the object code satisfies certain desirable properties [16, 18]. A separate component called the *verifier* examines the object code and certificate and determines whether the object code actually satisfies the properties. A wide range of properties can be verified, including memory safety (unallocated portions of memory are not accessed), control safety (code is entered only at valid entry points), and various security properties (e.g., highly classified data does appear on low security channels). Often, these properties are corollaries of type safety in an appropriate type system for the object code.

In this paper we will describe a certifying compiler for *Cyclone*, a high-level language that supports run-time code generation. Cyclone is compiled into *TAL/T*, an assembly language that supports run-time code generation. Cyclone and TAL/T are both type safe; the certificates of our system are the type annotations of the TAL/T output, and the verifier is the TAL/T type checker.

As compiler writers, we were motivated to implement Cyclone as a certifying compiler because we believe the approach enhances compiler correctness. For example, we were forced to develop a type system and operational semantics for TAL/T. This provides a formal framework for reasoning about object code that generates object code at run time. Eventually, we hope to prove that the compiler transforms type correct source programs into type correct object programs, an important step towards proving correctness for the compiler. In the meantime, we use the verifier to type check the output of the compiler, so that we get immediate feedback when our compiler introduces type errors. As others have noted [23, 24], this helps to identify and correct compiler bugs quickly.

We also wanted a certifying compiler to address the safety concerns of end users. In our system, type safety only depends on the certificate and the object code, and not on the method by which they are produced. Thus the end user does not have to rely on the programmer or the Cyclone compiler to ensure safety. This makes our system usable as the basis of security-critical applications like active networks and mobile code systems.

1.3 The Cyclone compiler

The Cyclone compiler is built on two existing systems, the Tempo specializer [19] and the Popcorn certifying compiler [16]. It has three phases, shown in Fig. 1.

The first phase transforms a type safe C program into a Cyclone program that uses run-time code generation. It starts by applying the static analyses of the Tempo system to a C program and context information that specifies which function arguments are invariant. The Tempo front end produces an action-annotated program. We added an additional pass to translate the action-annotated program into a Cyclone run-time specializer.

The second phase verifies that the Cyclone program is type safe, and then compiles it into TAL/T. To do this, we modified the Popcorn compiler of Morrisett et al.; Popcorn compiles a type safe dialect of C into TAL, a typed assembly language. We extended the front end of Popcorn to handle Cyclone programs, and modified its back end so that it outputs TAL/T. TAL/T is TAL extended with instructions for manipulating *templates*, code fragments parameterized by holes, and their corresponding types. This compilation phase not only transforms high-level Cyclone constructs into low-level assembly instructions, but also transforms Cyclone types into TAL/T types.

The third phase first verifies the type safety of the TAL/T program. The type system of TAL/T ensures that the templates are combined correctly and that holes are filled in correctly. This paper describes our overall approach and the front end in detail, but the details of TAL/T will appear in a subsequent paper. Finally, the TAL/T program is assembled and linked into an executable.

This three phase design offers a very flexible user interface since it allows programs to be written in C, Cyclone, or TAL/T. In the simplest case, the user can simply write a C program (or reuse an existing program) and allow the system to handle the rest. If the user desires more explicit control over the code generation process, he may write (or modify) a Cyclone program. If very fine-grain control is desired, the user can fine-tune a TAL/T program produced by Cyclone, or can write one by hand. Note that, since verification is performed at the TAL/T level, the same program safety properties are guaranteed in all three of these cases.

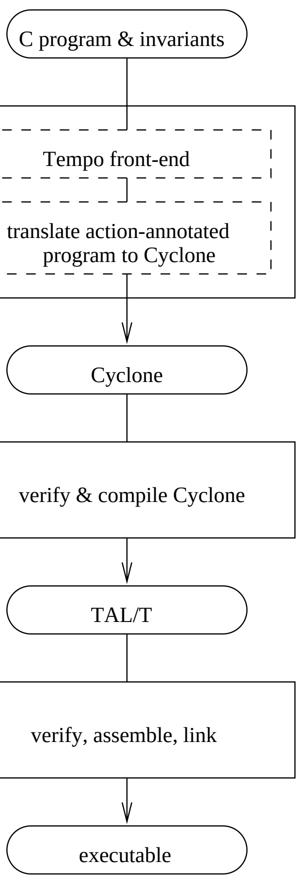

Figure 1: Overview of the Cyclone compiler

1.4 Example

We now present an example that illustrates run-time code generation and the phases of our Cyclone compiler. Fig. 2 shows a modular exponentiation function, `mexp`, written in standard C. Its arguments are a base value, an exponent, and a modulus. Modular exponentiation is often used in cryptography; when the same key is used to encrypt or decrypt several messages, the function is called repeatedly with the same exponent and modulus. Thus `mexp` can benefit from specialization.

To specialize the function with respect to a given exponent and modulus, the user indicates that the two arguments are *invariant*: the function will be called repeatedly with the same values for the invariant arguments. In Fig. 2, invariant arguments are shown in *italics*. A static analysis propagates this information throughout the program, producing an *action-annotated* program. Actions describe how each language construct will be treated during specialization. Constructs that depend only on invariants can be evaluated during specialization; these constructs are displayed in *italics* in the second part of the figure.

To understand how run-time specialization works, it is

C code (invariant arguments in *italics*)

```
int mexp(int base, int exp, int mod)
{
  int s, t, u;

  s = 1; t = base; u = exp;

  while (u != 0) {
    if ((u&1) != 0)
      s = (s*t) % mod;
    t = (t*t) % mod;
    u >>= 1;
  }
  return(s);
}
```

Action-annotated code (*italicized* constructs can be evaluated)

```
int mexp(int base, int exp, int mod)
{
  int s, t, u;

  s = 1; t = base; u = exp;

  while (u != 0) {
    if ((u & 1) != 0)
      s = (s*t) % mod;
    t = (t*t) % mod;
    u >>= 1;
  }
  return(s);
}
```

Specialized source code (`exp = 10`, `mod = 1234`)

```
int mexp_sp(int base)
{
  int s, t;

  s = 1; t = base;

  t = (t*t) % 1234;
  s = (s*t) % 1234;
  t = (t*t) % 1234;
  t = (t*t) % 1234;
  s = (s*t) % 1234;
  t = (t*t) % 1234;

  return(s);
}
```

Figure 2: Specialization at the source level

helpful to first consider how specialization could be achieved entirely within the source language. In our example, the specialized function `mexp_sp` of Fig. 2 is obtained from the action-annotated `mexp` when the exponent is 10 and the modulus is 1234. Italicized constructs of `mexp`, like the while loop, can be evaluated (note that the loop test depends only on the known arguments). Non-italicized constructs of `mexp` show up in the source code of `mexp_sp`. These constructs can only be evaluated when `mexp_sp` is called, because they depend on the unknown argument.

We can think of `mexp_sp` as being constructed by cutting and pasting together fragments of the source code of `mexp`. These fragments, or *templates*, are a central idea we used in designing Cyclone. Cyclone is a type safe dialect of C extended with four constructs that manipulate templates: `codegen`, `cut`, `splice`, and `fill`. Using these constructs, it is possible to write a Cyclone function that generates a specialized version of `mexp` at run time.

```
int (int) mexp_gen(int exp, int mod)
{
  int u;

  u = exp;

  return codegen(
    int mexp_sp(int base) {
    int s, t;

    s = 1; t = base;

    cut
      while (u != 0) {
        if ((u & 1) != 0)
          splice  s = (s*t) % fill(mod);
        splice  t = (t*t) % fill(mod);
        u >>= 1;
      }
    return(s);
    });
}
```

Figure 3: A run-time specializer written in Cyclone

In fact, our system can automatically generate a Cyclone run-time specializer from an action-annotated program. Fig. 3 shows the Cyclone specializer produced from the action-annotated modular exponentiation function of Fig. 2. The function `mexp_gen` takes the two invariant arguments of the original `mexp` function and returns the function `mexp_sp`, a version of `mexp` specialized to those arguments. In the figure we have italicized code that will be evaluated when `mexp_gen` is called. Non-italicized code is template code that will be manipulated by `mexp_gen` to produce the specialized function. The template code will only be evaluated when the specialized function is itself called.

In our example, the `codegen` expression begins the code generation process by allocating a region in memory for the new function `mexp_sp`, and copying the first template into the region. This template includes the declarations of the function, its argument `base`, and local variables `s` and `t`, and also the initial assignments to `s` and `t`. Recall that this template code is *not* evaluated during the code generation process, but merely manipulated.

The `cut` statement marks the end of the template and introduces code (italicized) that will be evaluated during code generation: namely, the while loop. The while test and body, including the conditional statement, `splice` statements, and shift/assignment statement, will all be evaluated. After the while loop finishes, the template following the `cut` statement (containing `return(s)`) will be added to the code generation region.

Evaluating a `splice` statement causes a template to be appended to the code generation region. In our example, each time the first `splice` statement is executed, an assignment to `s` is appended. Similarly, each time the second `splice` statement is executed, an assignment to `t` is appended. The effect of the while loop is thus to add some number of assignment statements to the code of `mexp_sp`; exactly how many, and which ones, is determined by the arguments of `mexp_gen`.

A `fill` expression can be used within a template, and it marks a *hole* in the template. When `fill(`e`)` is encountered in a template, e is evaluated at code generation time to a value, which is then used to fill the hole in the template. In our example, `fill` is used to insert the known modulus value into the assignment statements.

After code generation is complete, the newly generated function `mexp_sp` is returned as the result of `codegen`. It takes the one remaining argument of `mexp` to compute its result.

Cyclone programs can be evaluated symbolically to produce specialized source programs, like the one in Fig. 2; this is the basis of the formal operational semantics we give in the appendix. In our implementation, however, we compile Cyclone source code to object code, and we compile source templates into object templates. The Cyclone object code then manipulates object templates directly.

Our object code, TAL/T, is an extension of TAL with instructions for manipulating object templates. Most of the TAL/T instructions are x86 machine instructions; the new template instructions are `CGSTART`, `CGDUMP`, `CGFILL`, `CGHOLE`, `TEMPLATE_START`, and `TEMPLATE_END`. For example, the Cyclone program in Fig. 3 is compiled into the TAL/T program shown in Fig. 4. (We omitted some instructions to save space, and added source code fragments in comments to aid readability.)

The beginning of `_mexp_gen` contains x86 instructions for adding the local variable `u` to the stack and assigning it the value of the argument `exp`. Next, `CGSTART` is used to dynamically allocate a code generation region, and the first template is dumped (copied) into the region with the `CGDUMP` instruction. Next, the body of the loop is unrolled. Each Cyclone `splice` statement is compiled into a `CGDUMP` instruction, followed by instructions for computing hole values and a `CGFILL` instruction for filling in the hole. At the end of the `_mexp_gen` function, a final `CGDUMP` instruction outputs code for the last template.

Next comes the code for each of the four templates. The first template allocates stack space for local variables `s` and `t` and assigns values to them. The second and third templates come from the statements contained within the Cyclone `splice` instructions, i.e., the multiplications, mods, and assignments. The final template contains the code for `return(s)`. Each `CGHOLE` instruction introduces a placeholder inside a template, filled in during specialization as described above.

1.5 Summary

We designed a system for performing type safe run-time code generation. It has the following parts:

- C to action-annotated program translation
- Action-annotated program to Cyclone translation
- Cyclone language design
- Cyclone type system
- Cyclone verifier
- Cyclone to TAL/T compiler
- TAL/T language design
- TAL/T type system
- TAL/T verifier

```
_mexp_gen:
    PUSH    0                   ; int u = 0
    MOV     EAX,[ESP+8]         ; int u = exp;
    MOV     [ESP+0],EAX
    CGSTART                     ; codegen(...
    CGDUMP  ECX,cdgn_beg$18 ; (dump 1st template)
    JMP     whiletest$21        ; while ...
whilebody$20:
    :                           ; if ((u & 1) != 0)
    :
    CMP     EAX,ECX
    JE      ifend$24
    CGDUMP  ECX,splc_beg$25     ; (dump 2nd template)
    MOV     EAX,[ESP+12]        ; mod
    CGFILL  ECX,splc_beg$25,hole$29,EAX
ifend$24:
    CGDUMP  ECX,splc_beg$31     ; (dump 3rd template)
    MOV     EAX,[ESP+12]        ; mod
    CGFILL  ECX,splc_beg$26,hole$34,EAX
    MOV     ECX,1               ; u = u >> 1;
    MOV     EAX,[ESP+0]
    SAR     EAX,CL
    MOV     [ESP+0],EAX
whiletest$21:
    :
    :
    CMP     EAX,ECX             ; ... (u != 0)
    JNE     whilebody$20
whileend$22:
    CGDUMP  ECX,cut_beg$33      ; (dump 4th template)
    CGEND   EAX
    ADD     ESP,4               ; (return spec. fun.)
    RETN



TEMPLATE_START cdgn_beg$18,cdgn_beg$19
cdgn_beg$18                     ; (1st template)
    PUSH    0                   ; int s = 0;
    :
    MOV     EAX,[ESP+12]        ; t = base;
    MOV     [ESP+0],EAX
TEMPLATE_END cdgn_beg$19

TEMPLATE_START splc_beg$25,splc_end$26
splc_beg$25:                          ; (2nd template)
    CGHOLE  EAX,splc_beg$25,hole$29   ; fill(...)
    :
    MOV     [ESP+4],EAX               ; s = mod(s*t,...));
TEMPLATE_END splc_end$26

TEMPLATE_START splc_beg$31,splc_end$32
splc_beg$31:                          ; (3rd template)
    CGHOLE  EAX,splc_beg$31,hole$34   ; fill(...)
    :
    MOV     [ESP+0],EAX               ; t = mod(t*t,...));
TEMPLATE_END splc_end$32

TEMPLATE_START cut_beg$36,cut_end$37
cut_beg$36:                           ; (4th template)
    MOV     EAX,[ESP+4]               ; return s;
    ADD     ESP,8
    RETN
TEMPLATE_END cut_end$37
```

Figure 4: TAL/T code

- TAL/T to assembly translation
- Assembler/Linker

For some parts, we were able to reuse existing software. Specifically, we used Tempo for action-annotated program generation, Microsoft MASM for assembling, and Microsoft Visual C++ for linking. Other parts extend existing work. This was the case for the Cyclone language, type system, verifier, compiler, and the TAL/T language. Some components needed to be written from scratch, including the translation from an action-annotated program into a Cyclone program, and the definition of the new TAL/T instructions in terms of x86 instructions.

We've organized the rest of the paper as follows. In Section 2, we present the Cyclone language and its type system. In Section 3, we give a brief description of TAL/T; due to limited space we defer a full description to a later paper. We give implementation details and initial impressions about performance in Section 4. We discuss related work in Section 5, and future work in Section 6. Our final remarks are in Section 7.

2 Cyclone

2.1 Design decisions

Cyclone's `codegen`, `cut`, `splice`, and `fill` constructs were designed to express a template-based style of run-time code generation cleanly and concisely. We made some other design decisions based on Cyclone's relationship to the C programming language, and on implementation concerns.

First, because a run-time specializer is a function that returns a function as its result, we need higher order types in Cyclone. In C, higher order types can be written using pointer types, but Cyclone does not have pointers. Therefore, we introduce new notation for higher order types in Cyclone. For example:

```
int (float,int) f(int x) { ... }
```

This is a Cyclone function `f` that takes an int argument `x`, and returns a function taking a float and an int and returning an int. When `f` is declared and not defined, we use

```
int (int) (float,int) f;
```

Note that the type of the first argument appears to the left of the remaining arguments. This is consistent with the order the arguments would appear in C, using pointer types.

A second design decision concerns the extent to which we should support nested `codegen`'s. Consider the following example.

```
int (float) (int) f(int x) {
  return(codegen(
    int (int) g(float y) {
      return(codegen(
        int h(int z) {
        ... body of h ...
        }));
    }));
}
```

Here `f` is a function that generates a function `g` using `codegen` when called at run time. In turn, `g` will generate a function `h` each time it is called. Nested `codegen`'s are thus used to generate code that generates code. The first version of Tempo did not support code that generates code (though it has recently been extended to do so), and some other systems, such as 'C [20, 21], also prohibit it. We decided to permit it in Cyclone, because it adds little complication to our type system or implementation. Nested `codegen`'s are not generated automatically in Cyclone, because of the version of Tempo that we use, but the programmer can always write them explicitly.

A final design decision concerns the extent to which Cyclone should support lexically scoped bindings. In the last example, the function `h` is nested inside of two other functions, `f` and `g`. In a language with true lexical scoping, the arguments and local variables of these outer functions would be visible within the inner function: `f`, `x`, `g`, and `y` could be used in the body of `h`.

We decided that we would *not* support full lexical scoping in Cyclone. Our scoping rule is that in the body of a function, only the function itself, its arguments and local variables, and top-level variables are visible. This is in keeping with C's character as a low-level, machine- and systems-oriented language: the operators in the language are close to those provided by the machine, and the cost of executing a program is not hidden by high-level abstractions. We felt that closures and lambda lifting, the standard techniques for supporting lexical scoping, would stray too far from this. If lexical scoping is desired, the programmer can introduce explicit closures. Or, lexical scoping can be achieved using the Cyclone features, for example, if `y` is needed in the body of `h`, it can be accessed using `fill(y)`.

2.2 Syntax and typing rules

Now we formalize a core calculus of Cyclone. Full Cyclone has, in addition, structures, unions, arrays, void, break and continue, and for and do loops.

We use x to range over variables, c to range over constants, and b to range over base types. There is an implicit signature assigning types to constants, so that we can speak of "the type of c." Figure 5 gives the grammars for programs p, modifiers m, types t, declarations d, sequences D of declarations, function definitions F, statements s, and expressions e.

We write $t \bullet m$ for the type of a function from m to t: if $t = b\ m_1\ \cdots\ m_n$, then $t \bullet m = b\ m\ m_1\ \cdots\ m_n$. If $D = t_1\ x_1, \ldots, t_k\ x_n$, then $\widetilde{D}$ is defined to be the modifier $(t_1,\ \ldots\ ,\ t_n)$, so that a function definition $t\ x(D)\ s$ declares x to be of type $t \bullet \widetilde{D}$.

We sometimes consider a sequence $D = t_1\ x_1, \ldots, t_n\ x_n$ of declarations to be a finite function from variables to types: $D(x_i) = t_i$ if $1 \leq i \leq n$. This assumes that the x_i are distinct; we achieve this by alpha conversion when necessary, and by imposing some standard syntactic restrictions on Cyclone programs (the names of a function and its formal parameters must be distinct, and global variables have distinct names).

We define type environments E to support Cyclone's scoping rules:

$$\begin{array}{rcl} E & ::= & \texttt{outermost(}t\ x\texttt{(}D_{\text{params}}\texttt{);}\ D_{\text{local}}\texttt{);}\ D_{\text{global}} \\ & | & \texttt{frame(}t\ x\texttt{(}D_{\text{params}}\texttt{);}\ D_{\text{local}}\texttt{);}E \\ & | & \texttt{hidden(}t\ x\texttt{(}D_{\text{params}}\texttt{);}\ D_{\text{local}}\texttt{);}E \end{array}$$

Informally, a type environment is a sequence of hidden and visible *frames*, followed by an outermost frame that gives

$$E_{\text{vis}} = \begin{cases} D_2, D_1, t \bullet \widetilde{D_1}\ \ x, D_3 & \text{if } E = \texttt{outermost(}t\ \ x\texttt{(}D_1\texttt{);}\ \ D_2\texttt{);}\ \ D_3 \\ E'_{\text{vis}} & \text{if } E = \texttt{hidden(}t\ \ x\texttt{(}D\texttt{);}\ \ D'\texttt{)};E' \\ D', D, t \bullet \widetilde{D}\ \ x, E'_{\text{vis0}} & \text{if } E = \texttt{frame(}t\ \ x\texttt{(}D\texttt{);}\ \ D'\texttt{)};E' \end{cases}$$

$$E_{\text{vis0}} = \begin{cases} D_3 & \text{if } E = \texttt{outermost(}t\ \ x\texttt{(}D_1\texttt{);}\ \ D_2\texttt{);}\ \ D_3 \\ E'_{\text{vis0}} & \text{if } E = \texttt{hidden(}t\ \ x\texttt{(}D\texttt{);}\ \ D'\texttt{)};E' \\ E'_{\text{vis0}} & \text{if } E = \texttt{frame(}t\ \ x\texttt{(}D\texttt{);}\ \ D'\texttt{)};E' \end{cases}$$

$$\texttt{rtype}(E) = \begin{cases} t & \text{if } E = \texttt{outermost(}t\ \ x\texttt{(}D_1\texttt{);}\ \ D_2\texttt{);}\ \ D_3 \\ \texttt{rtype}(E') & \text{if } E = \texttt{hidden(}t\ \ x\texttt{(}D\texttt{);}\ \ D'\texttt{)};E' \\ t & \text{if } E = \texttt{frame(}t\ \ x\texttt{(}D\texttt{);}\ \ D'\texttt{)};E' \end{cases}$$

$$E + d = \begin{cases} \texttt{outermost(}t\ \ x\texttt{(}D_1\texttt{);}\ \ d\texttt{,}D_2\texttt{);}\ \ D_3 & \text{if } E = \texttt{outermost(}t\ \ x\texttt{(}D_1\texttt{);}\ \ D_2\texttt{);}\ \ D_3 \\ \texttt{hidden(}t\ \ x\texttt{(}D\texttt{);}\ \ D'\texttt{)};E' + d & \text{if } E = \texttt{hidden(}t\ \ x\texttt{(}D\texttt{);}\ \ D'\texttt{)};E' \\ \texttt{frame(}t\ \ x\texttt{(}D\texttt{);}\ \ d\texttt{,}D'\texttt{)};E' & \text{if } E = \texttt{frame(}t\ \ x\texttt{(}D\texttt{);}\ \ D'\texttt{)};E' \end{cases}$$

Figure 6: Cyclone environment functions

$$\begin{array}{lrcl}
\text{Programs} & p & ::= & \cdot \\
 & & | & d\texttt{;}\ \ p \\
 & & | & F\ \ p \\[1ex]
\text{Modifiers} & m & ::= & \texttt{(}t_1\texttt{,}\ \ \ldots\ \ \texttt{,}\ \ t_n\texttt{)} \\[1ex]
\text{Types} & t & ::= & b\ m_1\ \cdots\ m_n \\[1ex]
\text{Declarations} & d & ::= & t\ x \\[1ex]
\text{Decl. sequences} & D & ::= & d_1\texttt{,}\ \ \ldots\ \ \texttt{,}\ \ d_n \\[1ex]
\text{Function defns.} & F & ::= & t\ \ x\texttt{(}D\texttt{)}\ \ s \\[1ex]
\text{Statements} & s & ::= & e\texttt{;} \\
 & & | & d = e\texttt{;} \\
 & & | & \texttt{\{}\ \ s_1\ \ \cdots\ \ s_n\ \ \texttt{\}} \\
 & & | & \texttt{if (}e\texttt{)}\ \ s_1\ \ \texttt{else}\ \ s_2 \\
 & & | & \texttt{while (}e\texttt{)}\ \ s \\
 & & | & \texttt{return}\ \ e\texttt{;} \\
 & & | & \texttt{splice}\ \ s \\
 & & | & \texttt{cut}\ \ s \\[1ex]
\text{Expressions} & e & ::= & x \\
 & & | & c \\
 & & | & e_0\texttt{(}e_1\texttt{,}\ \ \ldots\texttt{,}e_n\texttt{)} \\
 & & | & x\ \texttt{=}\ e \\
 & & | & \texttt{codegen(}F\texttt{)} \\
 & & | & \texttt{fill(}e\texttt{)}
\end{array}$$

Figure 5: The grammar of core Cyclone

the type of a top level function, the types of its local variables, and the types of global variables. The non-outermost frames contain the type of a function that will be generated at run time, and types for the parameters and local variables of the function. If E is a type environment, we write E_{vis} for the *visible declarations* of E; E_{vis} is defined in Figure 6. Informally, the definition says that the declarations of the first non-hidden frame and the global declarations are visible, and all other declarations are not visible. Note that E_{vis} is a sequence of declarations, so we may write $E_{\text{vis}}(x)$ for the type of x in E.

Figure 6 also defines two other important operations on environments: $\texttt{rtype}(E)$ is the return type for the function of the first non-hidden frame, and $E + d$ is the environment obtained by adding declaration d to the local declarations of the first non-hidden frame.

The typing rules of Cyclone are given in Figure 7. The interesting rules are those for `codegen`, `cut`, `splice`, and `fill`.

A `codegen` expression starts the process of run time code generation. To type $\texttt{codegen(}t\ \ x\texttt{(}D\texttt{)}\ \ s\texttt{)}$ in an environment E, we type the body s of the function in an environment $\texttt{frame(}t\ \ x\texttt{(}D\texttt{);}\ \ \cdot\texttt{);}\ \ E$. This makes the function x and its parameters D visible in the body, while any enclosing function, parameters, and local variables will be hidden.

An expression $\texttt{fill(}e\texttt{)}$ should only appear within a template. Our typing rule ensures this by looking at the environment: it must have the form $\texttt{frame(}t\ \ x\texttt{(}D\texttt{);}\ \ D'\texttt{);}\ \ E$. If so, the expression $\texttt{fill(}e\texttt{)}$ is typed if e is typed in the environment $\texttt{hidden(}t\ \ x\texttt{(}D\texttt{);}\ \ D'\texttt{);}\ \ E$. That is, the function being generated with `codegen`, as well as its parameters and local variables, are hidden when computing the value that will fill the hole. This is necessary because the parameters and local variables will not become available until the function is called; they will not be available when the hole is filled.

The rules for `cut` and `splice` are similar. Like `fill`, `cut` can only be invoked within a template, and it changes `frame` to `hidden` for the same reason as `fill`. `Splice` is the dual of `cut`; it changes a frame `hidden` by `cut` back into a visible `frame`. Thus `splice` introduces a template, and `cut` interrupts a template.

An operational semantics for Cyclone and safety theorem are given in an appendix.

$$\boxed{D \vdash p} \qquad (p \text{ is a well-formed program})$$

$$D \vdash \cdot$$

$$\frac{D \vdash p}{D \vdash t\ x\texttt{;}\ p} \quad D(x) = t$$

$$\frac{D \vdash p, \quad D(x) = t \bullet \widetilde{D'} \qquad \texttt{outermost(}t\ x\texttt{(}D'\texttt{);}\ \cdot\ \texttt{);}\ D \vdash s}{D \vdash t\ x\texttt{(}D'\texttt{)}\ s\ p}$$

$$\boxed{E \vdash s} \qquad (s \text{ is a well-formed statement})$$

$$\frac{E \vdash e : t}{E \vdash e\texttt{;}}$$

$$\frac{E \vdash e : t}{E \vdash t\ x = e\texttt{;}}$$

$$\frac{}{E \vdash \texttt{\{ \}}}$$

$$\frac{E \vdash d = e\texttt{;} \quad E + d \vdash \texttt{\{}\ s_1\ \cdots\ s_n\ \texttt{\}}}{E \vdash \texttt{\{}\ d = e\texttt{;}\ s_1\ \cdots\ s_n\ \texttt{\}}}$$

$$\frac{E \vdash s_0,\ E \vdash \texttt{\{}\ s_1\ \cdots\ s_n\ \texttt{\}}}{E \vdash \texttt{\{}\ s_0\ s_1\ \cdots\ s_n\ \texttt{\}}} \quad s_0 \neq d = e\texttt{;}$$

$$\frac{E \vdash e : \texttt{int}, \quad E \vdash s_1, \quad E \vdash s_2}{E \vdash \texttt{if (}e\texttt{)}\ s_1\ \texttt{else}\ s_2}$$

$$\frac{E \vdash e : \texttt{int}, \quad E \vdash s}{E \vdash \texttt{while (}e\texttt{)}\ s}$$

$$\frac{E \vdash e : t}{E \vdash \texttt{return}\ e\texttt{;}} \quad \texttt{rtype}(E) = t$$

$$\frac{\texttt{frame(}t\ x\texttt{(}D\texttt{);}\ D'\texttt{)};E \vdash s}{\texttt{hidden(}t\ x\texttt{(}D\texttt{);}\ D'\texttt{)};E \vdash \texttt{splice}\ s}$$

$$\frac{\texttt{hidden(}t\ x\texttt{(}D\texttt{);}\ D'\texttt{)};E \vdash s}{\texttt{frame(}t\ x\texttt{(}D\texttt{);}\ D'\texttt{)};E \vdash \texttt{cut}\ s}$$

$$\boxed{E \vdash e : t} \qquad (e \text{ has type } t)$$

$$E \vdash x : t \qquad \text{if } E_{\text{vis}}(x) = t$$

$$E \vdash c : t \qquad \text{where } t \text{ is the type of the constant } c$$

$$\frac{E \vdash e_0 : t \bullet \texttt{(}t_1\texttt{,}\ \ldots\texttt{,}\ t_n\texttt{)},\ E \vdash e_1 : t_1\texttt{,}\ \ldots\texttt{,}\ E \vdash e_n : t_n}{E \vdash e_0\texttt{(}e_1\texttt{,}\ \ldots\texttt{,}e_n\texttt{)} : t}$$

$$\frac{E \vdash x : t,\ E \vdash e : t}{E \vdash x\ \texttt{=}\ e : t}$$

$$\frac{\texttt{frame(}t\ x\texttt{(}D\texttt{);}\ \cdot\texttt{)};E \vdash s}{E \vdash \texttt{codegen(}t\ x\texttt{(}D\texttt{)}\ s\texttt{)} : t \bullet \widetilde{D}}$$

$$\frac{\texttt{hidden(}t\ x\texttt{(}D\texttt{);}\ D'\texttt{)};E \vdash e : t}{\texttt{frame(}t\ x\texttt{(}D\texttt{);}\ D'\texttt{)};E \vdash \texttt{fill(}e\texttt{)} : t}$$

Figure 7: Typing rules of Cyclone

3 TAL/T

The output of the Cyclone compiler is a program in TAL/T, an extension of the Typed Assembly Language (TAL) of Morrisett et al. [16]. In designing TAL/T, our primary concern was to retain the low-level, assembly language character of TAL. Most TAL instructions are x86 machine instructions, possibly annotated with type information. The exceptions are a few macros, such as `malloc`, that would be difficult to type in their expanded form; each macro expands to a short sequence of x86 instructions. Since each instruction is simple, the trusted components of the system—the typing rules, the verifier, and the macros—are also simple. This gives us a high degree of confidence in the correctness and safety of the system.

TAL already has instructions that are powerful enough to generate code at run time: `malloc` and move are sufficient. The problem with this approach is in the types. If we `malloc` a region for code, what is its type? Clearly, by the end of the code generation process, it should have the type of TAL code that can be jumped to. But at the start of code generation, when it is not safe to jump to, it must have a different type. Moreover, the type of the region should change as we move instructions into it. The TAL type system is not powerful enough to show that a sequence of `malloc` and move instructions results in a TAL program that can safely be jumped to.

Our solution, TAL/T, is an extension of TAL with some types and macros for manipulating templates. Since this paper focuses on Cyclone and the front end of system, we will only sketch the ideas of TAL/T here. Full details will appear in a subsequent paper.

In TAL, a procedure is just the label or address of a sequence of TAL instructions. A procedure is called by jumping to the label or address. The type of a procedure is a precondition that says that on entry, the x86 registers should contain values of particular types. For example, if a procedure is to return it will have a precondition saying that a return address should be accessible through the stack pointer when it is jumped to.

In TAL/T, a template is also the label of a sequence of instructions. Unlike a TAL procedure, however, a template is not meant to be jumped to. For example, it might need to be concatenated with another template to form a TAL procedure. Thus the type of a template includes a postcondition as well as a precondition. Our typing rules for the template instructions of TAL/T will ensure that before a template is dumped into a code generation region, its precondition matches the postcondition of the previous template dumped. Also, a template may have holes that need to be filled; the types of these holes are also given in the type of the template.

The type of a code generation region is very similar to that of a template: it includes types for the holes that remain to be filled in the region, the precondition of the first template that was dumped, and the postcondition of the last template that was dumped. When all holes have been filled and a template with no postcondition is dumped, the region will have a type consisting of just a precondition, i.e., the type of a TAL procedure. At this point code generation is finished and the result can be jumped to.

```
int f(int x) {
  return(codegen(
    int g(int y) {
      return y+1;
    })(x));
}

int h(int x)(int) {
  return(codegen(
    int k(int y) {
      return(fill(f(x)))
    }));
}
```

Figure 8: An example showing that two `codegen` expressions can be executing at once. When called, `h` starts generating `k`, but stops in the middle to call `f` which generates `g`.

Now we give a brief description of the new TAL/T macros. This is intended to be an informal description showing that each macro does not go beyond what is already in TAL—the macros are low level, and remain close to machine code.

The macros manipulate an implicit stack of code generation regions. Each region in the stack is used for a function being generated by a `codegen`. The stack is needed because it is possible to have two `codegen` expressions executing at once (for an example, see Figure 8).

- `cgstart` initiates run-time code generation by allocating a new code generation region. This new region is pushed onto the stack of code generation regions and becomes the "current" region. The `cgstart` macro is about as complicated as `malloc`.
- `cgdump` r, L copies the template at label L into the current code generation region. After execution, the register r points to the copy of the template, and can be used to fill holes in the copy. `Cgdump` is our most complicated macro: its core is a simple string-copy loop, but it must also check that the current code generation region has enough room for a copy of the template. If there is not enough room, `cgdump` allocates a new region twice the size of the old region, copies the contents of the old region plus the new template to the new region, and replaces the old region with the new on the region stack. This is the most complex TAL/T instruction, consisting of roughly twenty x86 instructions.
- `cghole` r, L_{template}, L_{hole} is a move instruction containing a hole. It should be used in a template with label L_{template}, and declares the hole L_{hole}.
- `cgfill` r_1, L_{template}, L_{hole}, r_2 fills the hole of a template; it is a simple move instruction. Register r_1 should point to a copy of the template at label L_{template}, which should have a hole with label L_{hole}. Register r_2 contains the value to put in the hole.
- `cgfillrel` fills the hole of a template with a pointer into a second template; like `cgfill` it expands to a simple move instruction. It is needed for jumps between templates.

```
int f() {
  return(codegen(
    int g(int i) {
      cut { return 4; }
      return(i+1);
    })(7));
}
```

Figure 9: An example that shows the need for `cgabort`. When called, the function `f` starts generating function `g` but aborts in the middle (it returns 4).

- `cgabort` aborts a code generation; it pops the top region off the region stack. It is needed when the run-time code generation of a function stops in the middle, as in the example of Figure 9.
- `cgend` r finalizes the code generation process: the current region is popped off the region stack and put into register r. TAL can then jump to location r.

4 Implementation Status

We now describe some key aspects of our implementation. As previously mentioned, some components were written from scratch, while others were realized by modifying existing software.

4.1 Action-annotated program to Cyclone

We translate Tempo action-annotated programs into run-time specializers written in Cyclone. Using the Tempo front end, this lets us automatically generate a Cyclone program from a C program.

An action-annotated program distinguishes two kinds of code: normal code that will be executed during specialization, indicated in italics in Fig. 2; and template code that will emitted during specialization (non-italicized code). The annotated C program is translated into a Cyclone program that uses `codegen`, `cut`, `splice`, and `fill`. Since italicized constructs will be executed during code generation, they will occur outside `codegen`, or within a `cut` statement or a `fill` expression. Non-italicized constructs will be placed within a `codegen` expression or `splice` statement.

Our algorithm operates in two modes: "normal" mode translates constructs that should be executed at code generation time and "template" mode translates constructs that will be part of a template. The algorithm performs a recursive descent of the action-annotated abstract syntax, keeping track of which mode it is in. It starts off in "normal" mode and produces Cyclone code for the beginning of the run-time specializer: its arguments (the invariants) and any local variables and initial statements that are annotated with italics. When the first non-italic construct is encountered, a `codegen` expression is issued, putting the translation into "template" mode. The rest of the program is translated as follows.

An italic statement or expression must be translated in "normal" mode. Therefore, if the translation is in "template" mode, we insert `cut` (if we are processing a statement) or `fill` (if we are processing an expression) and switch into "normal" mode. Similarly, a non-italic statement should be translated in "template" mode; here we insert `splice` and switch modes if necessary. It isn't possible to encounter a non-italic expression within an italic expression.

Another step needs to be taken during this translation since specialization is *speculative*, i.e., both branches of a conditional statement can be optimistically specialized when the conditional test itself cannot be evaluated. This means that during specialization, the store needs to be saved prior to specializing one branch and restored before specializing the other branch. Therefore, we must introduce Cyclone statements to save and restore the store when translating such a conditional statement. This is the same solution used by Tempo [6].

4.2 Cyclone to TAL/T

To compile Cyclone to TAL/T, we extended an existing compiler, the Popcorn compiler of Morrisett et al. Popcorn is written in Caml, and it compiles a type safe dialect of C into TAL, a typed assembly language [16]. Currently, Popcorn is a very simple, stack based compiler, though it is being extended with register allocation and more sophisticated optimizations.

The Popcorn compiler works by performing a traversal of the abstract syntax tree, emitting TAL code as it goes. It uses an environment data structure of the following form:

```
type env = { local_env: (id * int) list;
             args_on_stack: int }
```

The environment maintains the execution state of each function as it is compiled. The field `local_env` contains each variable identifier and its corresponding stack offset. Arguments are pushed onto the stack prior to entry to the function body; the field `args_on_stack` records the number of arguments, so they can be popped off the stack upon exiting the function.

To compile Cyclone we needed to extend the environment datatype: first, because Cyclone switches between generating normal code and template code, and second, because Cyclone has nested functions. Therefore, we use environments with the same structure as the environments used in Cyclone's typing rules:

```
type cyclone_env =
    Outermost of env * (id list)
  | Frame of env * cyclone_env
  | Hidden of env * cyclone_env
```

That is, environments are sequences of type frames for functions. A frame can either be outermost, normal, or hidden. Once we have this type of environment, visible bindings are defined as they are for E_{vis} in Section 2.

An `Outermost` frame contains the local environment for a top-level function as well as the global identifiers. A `Frame` is used when compiling template code. A new `Frame` environment is created each time `codegen` is encountered. A `Frame` becomes `Hidden` to switch back to "normal" mode when a `cut` or `fill` is encountered.

Popcorn programs are compiled by traversing the abstract syntax tree and translating each Popcorn construct into the appropriate TAL instructions; the resulting sequence of TAL instructions is the compiled program. Compiling a Cyclone program, however, is more complicated; it is performed in two phases. The first phase alternates between generating normal and template TAL/T instructions and a second phase rearranges the instructions to put them in their proper place. In order for the instructions to be rearranged in the second phase, the first phase interleaves special *markers* with the TAL/T instructions:

```
type marker =
    M_TemplateBeg of id * id
  | M_TemplateEnd
  | M_Fill of id * exp
```

These markers are used to indicate which instructions are normal, which belong within a template, and which are used to fill holes. `M_TemplateBeg` takes two arguments, the beginning and ending label of a template, and is issued at the beginning of a template (when `codegen` or `splice` is encountered, or `cut` ends). Similarly, `M_TemplateEnd` is issued at the end of a template (at the end of a `codegen` or `splice`, or the beginning of a `cut`). Note that between corresponding `M_TemplateBeg` and `M_TemplateEnd` markers, other templates may begin and end. Therefore, these markers can be nested. When a hole is encountered, a `M_Fill` marker is issued. The first argument of `M_Fill` is a label for the hole inside the template. The second argument is the Cyclone source code expression that should fill the hole.

The following example shows how the `cut` statement is compiled.

```
fun compile_stmt stmt cyclone_env =
 match stmt of
  Cut s ->
   match cyclone_env with
     Outermost _ -> raise Error
   | Frame(env,cyclone_env') ->
       cg_fill_holes (Hidden(env,cyclone_env'));
       compile_stmt s (Hidden(env,cyclone_env'));
       emit_mark(M_TemplateBeg(id_new "a", id_new "b"));
   | Hidden _ -> raise Error
```

The function `compile_stmt` takes a Cyclone statement and an environment, and emits TAL/T instructions as a side-effect. The first thing to notice is that a cut can only occur when the compiler is in "template" mode, in which case the environment begins with `Frame`. A `cut` statement ends a template. Therefore, `cg_fill_holes` is called, which emits a `M_TemplateEnd` marker, and emits TAL/T code to dump the template and fill its holes. Filling holes must be done using a "normal" environment, and therefore the first frame becomes `Hidden`. Next, `compile_stmt` is called recursively to compile the statement `s` within the cut. Since the statement `s` should also be compiled in normal mode, it also keeps the first frame `Hidden`. Finally, a `M_TemplateBeg` marker is emitted so that the compilation of any constructs following the cut will occur within a new template.

The second phase of the code generation uses the markers to rearrange the code. The TAL/T instructions issued between a `M_TemplateBeg` and a `M_TemplateEnd` marker are extracted and made into a template. The remaining, normal instructions are concatenated to make one function; hole filling instructions are inserted after the instruction which dumps the template that contains the hole. The example in Fig. 4 shows a TAL/T program after the second phase is completed; the normal code includes instructions to dump templates and fill holes, and is followed by the templates.

4.3 TAL/T to executable

TAL is translated into assembly code by expanding each TAL macro into a sequence of x86 instructions. Similarly, the new TAL/T macros expand into a sequence of x86 and TAL instructions. A description of each TAL/T macro is given in Section 3. The resulting x86 assembly language program is assembled with Microsoft MASM and linked with the Microsoft Visual C++ linker.

4.4 Initial Impressions

We have implemented our system and have started testing it on programs to assess its strengths and weaknesses. Since there is currently a lot of interest in specializing interpreters, we decided to explore this type of application program. A state-of-the-art program specializer such as Tempo typically achieves a speedup between 2 and 20, depending on the interpreter and program interpreted. To see how our system compares, we took a bytecode interpreter available in the Tempo distribution and ran it through our system.

Preliminary results show that Cyclone achieves a speedup of over 3. This is encouraging, since this is roughly the speedup Tempo achieves on similar programs. A more precise comparison of the two systems still needs to be done, however. On the other hand, in our initial implementation, the cost of generating code is higher than in Tempo. One possible reason is that for safety, we allocate our code generation regions at run time, and perform bounds checks as we dump templates. The approach taken by Tempo, choosing a maximum buffer size at compile time and allocating a buffer of that size, is faster but not safe.

5 Related Work

Propagating types through all stages of a compiler, from the front end to the back end, has been shown to aid robust compiler construction: checking type safety after each stage quickly identifies compiler bugs [23, 24]. Additionally, Necula and Lee have shown that proving properties at the assembly language level is useful for safe execution of untrusted mobile code [18]. So far, this approach has been taken only for statically generated code. Our system is intended to achieve these same goals for dynamically generated code.

Many of the ideas in Cyclone were derived from the Tempo run-time specializer [7, 12, 13]. We designed Cyclone and TAL/T with a template-based approach in mind, and we use the Tempo front end for automatic template identification. Another run-time specializer, DyC, shares some of the same features, such as static analyses and a template-like back end [5, 9, 10]. There are, however, some important differences between Cyclone and these systems. We have tried to make our compiler more robust than Tempo and DyC, by making Cyclone type safe, and by using types to verify the safety of compiled code. Like Tempo and DyC, Cyclone can automatically construct specializers, but in addition, Cyclone also gives the programmer explicit control over run-time code generation, via the `codegen`, `cut`, `splice`, and `fill` constructs. It is even possible for us to hand-tweak the specializers produced by the Tempo front end with complete type safety. Like DyC, we can perform optimizations such as inter-template code motion, since we are writing our own compiler. Tempo's strategy of using an unmodified, existing compiler limits the optimizations that it can perform.

ML-box, Meta-ML, and 'C are all systems that add explicit code generation constructs to existing languages. ML-box and Meta-ML are type safe dialects of ML [15, 25, 22], while 'C is an unsafe dialect of C [20, 21]. All three systems have features for combining code fragments that go beyond what we provide in Cyclone. For example, in 'C it is possible to generate functions that have n arguments, where n is a value computed at run time; this is not possible in Cyclone, ML-box, or Meta-ML. On the other hand, 'C cannot generate a function that generates a function; this can be done in Cyclone (using nested `codegen`s), and also in ML-box and Meta-ML. An advantage we gain from not having sophisticated features for manipulating code fragments is simplicity: for example, the Cyclone type system does not need a new type for code fragments. The most fundamental difference, however, is that the overall system we present will provide type safety not only at the source level, but also at the object level. This makes our system more robust and makes it usable in a proof carrying code system.

6 Future Work

In this paper we presented a framework for performing safe and robust run-time code generation. Our compiler is based on a simple, stack-based, certifying compiler written by Morrisett et al. They are extending the compiler with register allocation and other standard optimizations, and we expect to merge Cyclone with their improvements.

We are interested in studying template-specific optimizations. For example, because templates appear explicitly in TAL/T, we plan to study inter-template optimizations, such as code motion between templates. Performing inter-template optimizations is more difficult in a system, like Tempo, based on an existing compiler that is not aware of templates.

We are also interested in analyses that could statically bound the size of the dynamic code generation region. This would let us allocate exactly the right amount of space when we begin generating code for a function, and would let us eliminate bounds checks during template dumps.

We would like to extend the front end of Tempo so that it takes Cyclone, and not just C, as input. This would mean extending the analyses of Tempo to handle Cyclone, which is an n-level language like ML-box. Additionally, we may implement the analysis of Glück and Jørgensen [8] to produce n-level Cyclone from C or Cyclone.

7 Conclusion

We have designed a programming language and compiler that combines dynamic code generation with certified compilation. Our system, Cyclone, has the following features.

Robust dynamic code generation Existing dynamic code generation systems only prove safety at the source level. Our approach extends this to object code. This means that bugs in the compiler that produce unsafe run-time specializers can be caught at compile time, before the specializer itself is run. This is extremely helpful because of the complexity of the analyses and transformations involved in dynamic code generation.

Flexibility and Safety Cyclone produces dynamic code generators that exploit run-time invariants to produce optimized programs. The user interface is flexible, since the final executable can be generated from a C program, a Cyclone program, or TAL/T assembly code. Type safety is statically verified in all three cases.

Safe execution of untrusted, dynamic, mobile code generators This approach can be used to extend a proof-carrying code system to include dynamic code generation. Since verification occurs prior to run time, there is no run-time cost incurred for the safety guarantees. Sophisticated optimization techniques can be employed in the certifying compiler. The resulting system could produce mobile code that is not only safe, but potentially extremely fast.

Acknowledgements. We were able to implement Cyclone quickly because we worked from the existing Tempo and TAL implementations. We'd like to thank Charles Consel and the Tempo group, and Greg Morrisett and the TAL group, for making this possible. The paper was improved by feedback from Julia Lawall and the anonymous referees.

References

[1] *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, Philadelphia, Pennsylvania, 21–24 May 1996. *SIGPLAN Notices* 31(5), May 1996.

[2] *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Amsterdam, The Netherlands, 12–13 June 1997.

[3] *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, Montreal, Canada, 17–19 June 1998. *SIGPLAN Notices* 33(5), May 1998.

[4] L. Augustsson. Partial evaluation in aircraft crew planning. In ACM [2], pages 127–136.

[5] J. Auslander, M. Philipose, C. Chambers, S. J. Eggers, and B. N. Bershad. Fast, effective dynamic compilation. In ACM [1], pages 149–159. *SIGPLAN Notices* 31(5), May 1996.

[6] C. Consel, L. Hornof, F. Noël, J. Noyé, and E. Volanschi. A uniform approach for compile-time and run-time specialization. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar, Dagstuhl Castle*, number 1110 in Lecture Notes in Computer Science, pages 54–72, Feb. 1996.

[7] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In *Conference Record of POPL '96: The 23$^{\mathrm{rd}}$ ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 145–156, St. Petersburg Beach, Florida, 21–24 Jan. 1996.

[8] R. Glück and J. Jørgensen. Fast binding-time analysis for multi-level specialization. In D. Bjørner, M. Broy, and I. V. Pottosin, editors, *Perspectives of System Informatics. Proceedings*, volume 1181 of *Lecture Notes in Computer Science*, pages 261–272. Springer-Verlag, 1996.

[9] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. DyC: An expressive annotation-directed dynamic compiler for C. *Theoretical Comput. Sci.* To appear.

[10] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. Annotation-directed run-time specialization in C. In ACM [2], pages 163–178.

[11] B. Guenter, T. Knoblock, and E. Ruf. Specializing shaders. In *Computer Graphics Proceedings*, Annual Conference Series, pages 343–350. ACM Press, 1995.

[12] L. Hornof. *Static Analyses for the Effective Specialization of Realistic Applications.* PhD thesis, Université de Rennes I, June 1997.

[13] L. Hornof and J. Noyé. Accurate binding-time analysis for imperative languages: Flow, context, and return sensitivity. In ACM [2], pages 63–73.

[14] D. Keppel, S. Eggers, and R. Henry. A case for runtime code generation. Technical Report 91-11-04, Department of Computer Science, University of Washington, Seattle, WA, 1991.

[15] M. Leone and P. Lee. Lightweight run-time code generation. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 97–106, Orlando, Florida, 25 June 1994. University of Melbourne, Australia, Department of Computer Science, Technical Report 94/9.

[16] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85–97, San Diego, California, 19–21 Jan. 1998.

[17] G. Muller, R. Marlet, E. Volanschi, C. Consel, C. Pu, and A. Goel. Fast, optimized Sun RPC using automatic program specialization. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, Amsterdam, The Netherlands, May 1998. IEEE Computer Society Press.

[18] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In ACM [3], pages 333–344. *SIGPLAN Notices* 33(5), May 1998.

[19] F. Noël, L. Hornof, C. Consel, and J. Lawall. Automatic, template-based run-time specialization : Implementation and experimental study. In *International Conference on Computer Languages*, pages 132–142, Chicago, IL, May 1998. IEEE Computer Society Press. Also available as IRISA report PI-1065.

[20] M. Poletto, D. R. Engler, and M. F. Kaashoek. tcc: A system for fast, flexible, and high-level dynamic code generation. In *Proceedings of the ACM SIGPLAN'97 Conference on Programming Language Design and Implementation (PLDI)*, pages 109–121, Las Vegas, Nevada, 15–18 June 1997. *SIGPLAN Notices* 32(5), May 1997.

[21] M. Poletto, W. C. Hsieh, D. R. Engler, and M. F. Kaashoek. 'C and tcc: A language and compiler for dynamic code generation. *ACM Trans. Prog. Lang. Syst.* To appear.

[22] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In ACM [2], pages 203–217.

[23] D. Tarditi. *Design and Implementation of Code Optimizations for a Type-Directed Compiler for Standard ML*. PhD thesis, Carnegie Mellon University, Dec. 1997. CMU–CS–97–108.

[24] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In ACM [1], pages 181–192. *SIGPLAN Notices* 31(5), May 1996.

[25] P. Wickline, P. Lee, and F. Pfenning. Run-time code generation and Modal-ML. In ACM [3], pages 224–235. *SIGPLAN Notices* 33(5), May 1998.

A Operational semantics of Cyclone

In this appendix, we define a small-step operational semantics for Cyclone based on evaluation contexts.

It is convenient to use a modified grammar for Cyclone in the operational semantics; this grammar is summarized in Figure 10. We have extended the Cyclone expressions with a new form, $\langle\!\langle s\rangle\!\rangle_t$. This is an expression consisting of a statement that should finish computing by evaluating `return` on a value of type t. We have introduced a new class, f, of variables called *function names*. Function names are distinct from the usual variables, x, and are intended to be used in the semantics only—they are not present in user programs. *Values*, v, are constants or function names, but not variables x. *Heaps*, H, are sequences of definitions, for both normal variables and for function names. The variables defined in a heap must be distinct; in our semantics we implicitly use alpha conversion when necessary to achieve this. If a heap H defines variables $z_1, \ldots, z_n$ with types $t_1, \ldots, t_n$, then $\widetilde{H}$ is defined to be the sequence of declarations $t_1\ z_1, \ldots, t_n\ z_n$. It is sometimes convenient to use $\widetilde{H}$ where we would use D, although, strictly speaking, some of the z_i will be function names, contrary to the definition of D.

Our new syntax requires some new typing rules, presented below. We assume that there is a distinguished function name, `main`, and for declarations D we define $\hat{D}$ to be the environment `outermost(int main(); ·); D`. The name `main` is not permitted anywhere else. Additionally, we extend Cyclone type environments as follows:

$$E ::= \cdots \mid \texttt{return}(t)\texttt{;}\ E$$

This marks an expected return type, and is used in typing the new construct, $\langle\!\langle s\rangle\!\rangle_t$. The environment functions, like $\texttt{rtype}(E)$, are extended in the obvious way.

$\boxed{E \vdash e : t}$ (e has type t)

$$\frac{\texttt{return}(t)\texttt{;}\ E \vdash s}{E \vdash \langle\!\langle s\rangle\!\rangle_t : t}$$

$$E \vdash f : t \quad \text{if } E_{\text{vis}}(f) = t$$

$\boxed{D \vdash H}$ (H is a well-formed heap)

$$D \vdash \cdot$$

$$\frac{\hat{D} \vdash v : t, \quad D \vdash H}{D \vdash H, d = v} \quad d = (t\ x) \in D$$

$$\frac{\texttt{outermost}(t\ \ f(D')\texttt{;}\ \cdot\ )\texttt{;}\ D \vdash s \qquad D \vdash H}{D \vdash H, t\ \ f(D')\ \ s} \quad \text{where } D(f) = t \bullet \widetilde{D'}$$

$\boxed{\vdash H, s}$ (H, s is a well-formed program)

$$\frac{\widetilde{H} \vdash H, \quad \hat{\widetilde{H}} \vdash s}{\vdash H, s}$$

The evaluation contexts for Cyclone are given in Figure 11. We need about four times the number of contexts as usual, because Cyclone distinguishes statements from expressions, and because we have both normal evaluation and evaluation under templates. For example, the context SE[·] becomes a statement when its hole is filled by an expression, and the context SS[·] becomes a statement when its hole is filled by a statement. Similarly, the context TES[·] becomes an expression when its hole is filled by a statement; the "T" contexts are meant to be used within templates. Fortunately, the variations between the different classes of evaluation context are small.

Our rewriting semantics is defined by identifying "redexes," which are the syntactic subterms where rewriting will take place. We have two kinds of redexes, expression redexes and statement redexes. The expression redexes are:

- x
- $v_0\texttt{(}v_1, \ldots, v_n\texttt{)}$
- $x = v$
- $\langle\!\langle\{\}\rangle\!\rangle_t$
- $\texttt{codegen(}t\ \ x(D)\ \ \overline{s}\texttt{)}$
- $\texttt{fill(}v\texttt{)}$

The statement redexes are:

- $v\texttt{;}$
- $d = v\texttt{;}$
- $\{\ \{\}\ \ s \cdots\ \}$
- $\texttt{if}\ \ (v)\ \ s_1\ \ s_2$
- $\texttt{while}\ \ (e)\ \ s$
- $\texttt{return}\ \ v\texttt{;}$
- $\texttt{splice}\ \ s$
- $\texttt{cut}\ \ \{\}$

The evaluation contexts are used to specify which redex in a program should be evaluated first; that is, they give the evaluation order. The following lemma (easily proved by induction) shows that our evaluation contexts and redexes exactly capture all of the cases we will need to consider to define a complete, deterministic evaluation relation.

Lemma. For any statement s, exactly one of the following cases holds:

- there is a unique context SS[·] such that $s = \text{SS}[s']$, where s' is a statement redex;
- there is a unique context SE[·] such that $s = \text{SE}[e]$, where e is an expression redex; or

$$\begin{array}{rcl@{\qquad}rcl}
m & ::= & (t_1,\ \ldots\ ,\ t_n) & t & ::= & b\ m_1\ \cdots\ m_n \\
d & ::= & t\ x & D & ::= & d_1, \ldots, d_n \\
v & ::= & c \mid f & H & ::= & \cdot \mid H, d = v \mid H, t\ \ f(D)\ \ s \\
s & ::= & e; & \overline{s} & ::= & \overline{e}; \\
 & \mid & d = e; & & \mid & d = \overline{e}; \\
 & \mid & \{\ s_1\ \cdots\ s_n\ \} & & \mid & \{\ \overline{s_1}\ \cdots\ \overline{s_n}\ \} \\
 & \mid & \texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2 & & \mid & \texttt{if}\ (\overline{e})\ \overline{s_1}\ \texttt{else}\ \overline{s_2} \\
 & \mid & \texttt{while}\ (e)\ s & & \mid & \texttt{while}\ (\overline{e})\ \overline{s} \\
 & \mid & \texttt{return}\ e; & & \mid & \texttt{return}\ \overline{e}; \\
 & \mid & \texttt{splice}\ s & & \mid & \texttt{splice}\ \overline{s} \\
 & \mid & \texttt{cut}\ s & & & \\
e & ::= & x & \overline{e} & ::= & x \\
 & \mid & c & & \mid & c \\
 & \mid & f & & \mid & f \\
 & \mid & e_0(e_1,\ \ldots, e_n) & & \mid & \overline{e_0}(\overline{e_1},\ \ldots, \overline{e_n}) \\
 & \mid & x\ \texttt{=}\ e & & \mid & x\ \texttt{=}\ \overline{e} \\
 & \mid & \langle\!\langle s \rangle\!\rangle_t & & \mid & \langle\!\langle \overline{s} \rangle\!\rangle_t \\
 & \mid & \texttt{codegen(}t\ \ x(D)\ \ s\texttt{)} & & \mid & \texttt{codegen(}t\ \ x(D)\ \ s\texttt{)} \\
 & \mid & \texttt{fill(}e\texttt{)} & & &
\end{array}$$

Figure 10: Grammars for Cyclone's operational semantics

$$\begin{array}{rcl@{\qquad}rcl}
\mathrm{SE}[\cdot] & ::= & & \mathrm{SS}[\cdot] & ::= & [\cdot] \\
 & & \mathrm{EE}[\cdot]; & & \mid & \mathrm{ES}[\cdot]; \\
 & \mid & d = \mathrm{EE}[\cdot]; & & \mid & d = \mathrm{ES}[\cdot]; \\
 & \mid & \{\ \mathrm{SE}[\cdot]\ s_1 \cdots s_n\ \} & & \mid & \{\ \mathrm{SS}[\cdot]\ s_1 \cdots s_n\ \} \\
 & \mid & \texttt{if}\ (\mathrm{EE}[\cdot])\ s_1\ \ s_2 & & \mid & \texttt{if}\ (\mathrm{ES}[\cdot])\ s_1\ \ s_2 \\
 & \mid & \texttt{return}\ \mathrm{EE}[\cdot]; & & \mid & \texttt{return}\ \mathrm{ES}[\cdot]; \\
 & \mid & \texttt{cut}\ \mathrm{SE}[\cdot] & & \mid & \texttt{cut}\ \mathrm{SS}[\cdot] \\
\mathrm{EE}[\cdot] & ::= & [\cdot] & \mathrm{ES}[\cdot] & ::= & \\
 & \mid & \mathrm{EE}[\cdot](e_1\ ,\ \ldots\ ,\ e_n) & & & \mathrm{ES}[\cdot](e_1\ ,\ \ldots\ ,\ e_n) \\
 & \mid & v(v',\ \ldots\ ,\ \mathrm{EE}[\cdot]\ ,\ e,\ \ldots) & & \mid & v(v',\ \ldots\ ,\ \mathrm{ES}[\cdot]\ ,\ e,\ \ldots) \\
 & \mid & x = \mathrm{EE}[\cdot] & & \mid & x = \mathrm{ES}[\cdot] \\
 & \mid & \langle\!\langle \mathrm{SE}[\cdot] \rangle\!\rangle_t & & \mid & \langle\!\langle \mathrm{SS}[\cdot] \rangle\!\rangle_t \\
 & \mid & \texttt{codegen}\ (t\ \ x(D)\ \ \mathrm{TSE}[\cdot]) & & \mid & \texttt{codegen}\ (t\ \ x(D)\ \ \mathrm{TSS}[\cdot]) \\
 & \mid & \texttt{fill(}\mathrm{EE}[\cdot]\texttt{)} & & \mid & \texttt{fill(}\mathrm{ES}[\cdot]\texttt{)} \\
\mathrm{TSE}[\cdot] & ::= & \mathrm{TEE}[\cdot]; & \mathrm{TSS}[\cdot] & ::= & \mathrm{TES}[\cdot]; \\
 & \mid & d = \mathrm{TEE}[\cdot]; & & \mid & d = \mathrm{TES}[\cdot]; \\
 & \mid & \{\ \overline{s}\ \cdots\ \mathrm{TSE}[\cdot]\ s'\ \cdots\ \} & & \mid & \{\ \overline{s}\ \cdots\ \mathrm{TSS}[\cdot]\ s'\ \cdots\ \} \\
 & \mid & \texttt{if}\ (\mathrm{TEE}[\cdot])\ s_1\ \ s_2 & & \mid & \texttt{if}\ (\mathrm{TES}[\cdot])\ s_1\ \ s_2 \\
 & \mid & \texttt{if}\ (\overline{e})\ \mathrm{TSE}[\cdot]\ s_2 & & \mid & \texttt{if}\ (\overline{e})\ \mathrm{TSS}[\cdot]\ s_2 \\
 & \mid & \texttt{if}\ (\overline{e})\ \overline{s}\ \mathrm{TSE}[\cdot] & & \mid & \texttt{if}\ (\overline{e})\ \overline{s}\ \mathrm{TSS}[\cdot] \\
 & \mid & \texttt{while}\ (\mathrm{TEE}[\cdot])\ s & & \mid & \texttt{while}\ (\mathrm{TES}[\cdot])\ s \\
 & \mid & \texttt{while}\ (\overline{e})\ \mathrm{TSE}[\cdot] & & \mid & \texttt{while}\ (\overline{e})\ \mathrm{TSS}[\cdot] \\
 & \mid & \texttt{return}\ \mathrm{TEE}[\cdot]; & & \mid & \texttt{return}\ \mathrm{TES}[\cdot]; \\
 & \mid & \texttt{cut}\ \mathrm{SE}[\cdot] & & \mid & \texttt{cut}\ \mathrm{SS}[\cdot] \\
\mathrm{TEE}[\cdot] & ::= & \mathrm{TEE}[\cdot](e_1\ ,\ \ldots\ ,\ e_n) & \mathrm{TES}[\cdot] & ::= & \mathrm{TES}[\cdot](e_1\ ,\ \ldots\ ,\ e_n) \\
 & \mid & \overline{e_0}(\overline{e_1},\ \ldots\ ,\ \mathrm{TEE}[\cdot]\ ,\ e_n,\ \ldots) & & \mid & \overline{e_0}(\overline{e_1},\ \ldots\ ,\ \mathrm{TES}[\cdot]\ ,\ e_n,\ \ldots) \\
 & \mid & x = \mathrm{TEE}[\cdot] & & \mid & x = \mathrm{TES}[\cdot] \\
 & \mid & \langle\!\langle \mathrm{TSE}[\cdot] \rangle\!\rangle_t & & \mid & \langle\!\langle \mathrm{TSS}[\cdot] \rangle\!\rangle_t \\
 & \mid & \texttt{fill(}\mathrm{EE}[\cdot]\texttt{)} & & \mid & \texttt{fill(}\mathrm{ES}[\cdot]\texttt{)}
\end{array}$$

Figure 11: Evaluation contexts for Cyclone's operational semantics

- $s = \{\}$.

The constructs `return`, `cut`, `splice`, and `fill` may result in errors depending on the context in which they execute. The following definition makes the relevant conditions on the contexts precise.

Definition.

- A context $\mathrm{SS}[\cdot]$ is in *return mode* if its hole does not occur within $\langle\!\langle \cdot \rangle\!\rangle_t$.
- A context $\mathrm{SS}[\cdot]$ is in *codegen mode* if
$$\mathrm{SS}[\cdot] = \mathrm{SE}[\texttt{codegen(}t\ \ x\texttt{(}D\texttt{)}\ \ \mathrm{SS}'[\cdot]\texttt{)}]$$
where the hole of $\mathrm{SS}'[\cdot]$ does not occur within `cut`.
- A context $\mathrm{SE}[\cdot]$ is in *codegen mode* if
$$\mathrm{SE}[\cdot] = \mathrm{SE}_1[\texttt{codegen(}t\ \ x\texttt{(}D\texttt{)}\ \ \mathrm{SE}_2[\cdot]\texttt{)}]$$
where the hole of $\mathrm{SE}_2[\cdot]$ does not occur within `cut`.

Finally, the rewriting relation $\rightarrow$ is defined by the rewrite rules in Figure 12. The relation rewrites a heap and statement H, s. By the previous lemma and a case analysis of the rewrite rules, we can see that there are four disjoint possibilities:

- $H, s \rightarrow H', s'$ for some heap and statement H', s';
- $H, s \rightarrow \text{error}$;
- $H, s \rightarrow \text{error}_{\text{return}}$; or
- s is the statement $\{\}$, in which case evaluation is halted (there is no H', s' such that $H, s \rightarrow H', s'$, and H, s does not rewrite to error or $\text{error}_{\text{return}}$).

There are two kinds of error. Most errors are prevented by the type system; these are simply lumped together under "error." These errors include applying a function to the wrong number of arguments, trying to apply a non-function, using undefined variables, etc. There is also an error, "$\text{error}_{\text{return}}$," that is not prevented by our type system. It occurs when a function completes without executing a `return` statement. In our implementation, when this happens execution halts and the error is reported to the user. The full implementation has a few more errors of this sort, including array bound, stack overflow, and out of memory errors.

The safety theorem is stated as follows.

Theorem (Safety). If $\vdash H, s$, then there is no H', s' such that $H, s \rightarrow^* H', s' \rightarrow \text{error}$.

The theorem is proved by showing that types are preserved by each rule rewriting $H, s \rightarrow H', s'$, and by showing that if a rule rewrites $H, s \rightarrow \text{error}$, then H, s is not well-typed.

$$H, \mathrm{SE}[x] \;\rightarrow\; \begin{cases} H, \mathrm{SE}[v] & \text{if } H = H_1, t\ x = v, H_2 \\ \text{error} & \text{otherwise} \end{cases}$$

$$H, \mathrm{SE}[v_0\texttt{(}v_1\texttt{,}\ \ldots\texttt{,}\ v_n\texttt{)}] \;\rightarrow\; \begin{cases} H, \mathrm{SE}[\langle\!\langle \{d_1 = v_1\texttt{;}\ \cdots\texttt{;}\ d_n = v_n\texttt{;}\ s\} \rangle\!\rangle_t] \\ \qquad \text{if } v_0 = f \text{ and } H = H_1, t\ \ f\texttt{(}d_1, \ldots, d_n\texttt{)}\ \ s, H_2 \\ \text{error} \quad \text{otherwise} \end{cases}$$

$$H, \mathrm{SE}[x = v] \;\rightarrow\; \begin{cases} H_1, t\ x = v, H_2, \mathrm{SE}[v] & \text{if } H = H_1, t\ x = v', H_2 \\ \text{error} & \text{otherwise} \end{cases}$$

$$H, \mathrm{SE}[\langle\!\langle \{\} \rangle\!\rangle_t] \;\rightarrow\; \text{error}_{\text{return}}$$

$$H, \mathrm{SE}[\texttt{codegen(}t\ \ x\texttt{(}D\texttt{)}\ \ \overline{s}\texttt{)}] \;\rightarrow\; H, t\ \ f(D)\ \ \overline{s}[x := f], \mathrm{SE}[f] \quad \text{where } f \text{ is a fresh function name}$$

$$H, \mathrm{SE}[\texttt{fill(}v\texttt{)}] \;\rightarrow\; \begin{cases} H, \mathrm{SE}[v] & \text{where } \mathrm{SE}[\cdot] \text{ is in codegen mode} \\ \text{error} & \text{otherwise} \end{cases}$$

$$H, \mathrm{SS}[v\texttt{;}] \;\rightarrow\; H, \mathrm{SS}[\{\}]$$

$$H, \mathrm{SS}[d = v\texttt{;}] \;\rightarrow\; \begin{array}{l} H, d = v, \mathrm{SS}[\{\}] \\ \text{where the variable of } d \text{ does not appear in } H \end{array}$$

$$H, \mathrm{SS}[\{\ \ \{\}\ \ s \cdots\ \}] \;\rightarrow\; H, \mathrm{SS}[\{\ \ s \cdots\ \}]$$

$$H, \mathrm{SS}[\texttt{if (}v\texttt{)}\ \ s_1\ \ s_2] \;\rightarrow\; \begin{cases} H, \mathrm{SS}[s_2] & \text{if } v = 0 \\ H, \mathrm{SS}[s_1] & \text{otherwise} \end{cases}$$

$$H, \mathrm{SS}[\texttt{while (}e\texttt{)}\ \ s] \;\rightarrow\; H, \mathrm{SS}[\texttt{if (}e\texttt{)}\ \{\ s\ \ \texttt{while (}e\texttt{)}\ \ s\ \}\ \{\}]$$

$$H, \mathrm{SS}[\texttt{return}\ \ v\texttt{;}] \;\rightarrow\; \begin{cases} H, \mathrm{SE}[v] & \text{if } \mathrm{SS}[\cdot] = \mathrm{SE}[\langle\!\langle \mathrm{SS}'[\cdot] \rangle\!\rangle_t], \text{ and } \mathrm{SS}'[\cdot] \text{ is in return mode} \\ \text{error}_{\text{return}} & \text{otherwise} \end{cases}$$

$$H, \mathrm{SS}[\texttt{splice}\ \ s] \;\rightarrow\; \begin{cases} H, \mathrm{SS}_1[\{\ \ s\ \ \texttt{cut}\ \ \mathrm{SS}_2[\{\}]\ \ \}] & \text{if } \mathrm{SS}[\cdot] = \mathrm{SS}_1[\texttt{cut}\ \ \mathrm{SS}_2[\cdot]], \\ & \quad \text{where } \mathrm{SS}_1[\cdot] \text{ is in codegen mode} \\ \text{error} & \text{otherwise} \end{cases}$$

$$H, \mathrm{SS}[\texttt{cut}\ \ \{\}] \;\rightarrow\; \begin{cases} H, \mathrm{SS}[\{\}] & \text{if the hole of } \mathrm{SS}[\cdot] \text{ is in codegen mode} \\ \text{error} & \text{otherwise} \end{cases}$$

Figure 12: Rewrite rules of Cyclone's operational semantics