CERTIFIED RUN-TIME CODE GENERATION

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Frederick Mattsson Smith

August 2001

CERTIFIED RUN-TIME CODE GENERATION

Frederick Mattsson Smith, Ph.D.

Cornell University 2001

Run-time code generation (RTCG) has been shown to be an effective optimization. Systems such as DyC, 'C, Tempo, and Fabius have demonstrated order of magnitude improvements on benchmark programs. In addition, by using binding time analyses, the benefits of RTCG can be obtained with minimal programmer effort. Languages like MetaML and PML have explored type systems for RTCG that can ensure programs using RTCG do not crash. In earlier work we presented Cyclone – a type-safe language based on C with support for RTCG.

The safety guarantee provided by a type-safe source language may fail to hold of the compiler's output. Rather than trusting that the compiler is correctly implemented, we can use a certifying compiler to check that the output is type-safe. A certifying compiler produces a certificate, along with the object code, which together can be used to verify that the object code is safe.

Here we develop a certifying and *optimizing* compiler for Cyclone supporting run-time code generation. The output of our compiler is TAL/T – x86 object code augmented with typing annotations. Part of our system is a verifier for TAL/T capable of mechanically checking that TAL/T programs are type safe – they cannot crash. This guarantee provides increased confidence in the compiler-generated code.

The contributions of this thesis include: 1) a framework for compiling and opti-

mizing programs supporting RTCG, 2) a formal proof that well-typed TAL/T programs cannot crash, and 3) an evaluation of our compiler using micro-benchmarks, a raytracer, and an instruction set simulator.

Biographical Sketch

Frederick Mattsson Smith was born on July 17,1973 in São Paulo, Brazil to Dorrit Mattsson Smith and Willard Leeds Smith. At one year of age, he departed Brazil for Düsseldorf, Germany. In 1976, excited by the Portuguese revolution, he moved there. The revolution proved uneventful but charmed by the people, the climate, the wine, and the great bread; Frederick decided to remain. In 1990, he left his parents, his siblings (Martin Åke Smith and Emilia Mabel Smith), and Portugal to attend Vanderbilt University in Nashville, Tennessee. In 1995 he completed his Bachelors degree in Physics and Math. After that he decided to pursue gradutat studies in Computer Science at Cornell University. In September 2001, having completed all requirements, Frederick received his Ph.D.

Acknowledgements

Thank you all. Its always good to be specific.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Efficient and reliable software is hard to develop. This thesis presents and evaluates a technology for combining run-time code generation and certifying compilation designed to simplify this process.

Run-time code generation (RTCG), also known as dynamic code generation, broadly encompasses any technique permitting a program to generate specialized code at run time. For the purpose of this thesis, we use RTCG for tools that help programmers make use of dynamic code generation.

Programs benefit from generating code at run time by exploiting run-time invariants. For example, inputs to a program can be used to eliminate unnecessary tests, unroll loops, embed constants in the instruction stream. In general, any data-driven compile-time optimization can be extended into run time through this technique. The benefits gained by RTCG will be offset by the time to generate the dynamic code.

A number of tools have shown RTCG to be a very effective optimization. With a tool like Fabius, DyC, or Tempo; minimal programmer annotations drive the application of RTCG. Other systems such as 'C and MetaML require greater programmer intervention but provide greater control. Small numeric kernels often run an order of magnitude faster than their unoptimized counterparts [NHCL96] using these techniques. For larger applications like simulators and remote procedure call, previous work has shown speedups of around a factor of two [GMP$^+$00].

'C, Fabius, Tempo, and DyC have developed techniques to minimize the cost of code generation without sacrificing code quality. These lightweight approaches to RTCG can produce machine code between ten and one hundred times faster than a traditional compiler [PHEK99, NHCL96]. These speed gains are obtained by either using fast linear optimization techniques or by optimizing the functions that will be dynamically generated at compile time.

As shown by this previous work, a tool supporting RTCG can help programmers achieve significant speedups with minimal effort. This combination of simplicity and effectiveness originally attracted me to this area.

One drawback of RTCG is that it is untested technology. Compilers are hard to write and optimizations are hard to reason about even without RTCG. Debuggers and profilers do not support RTCG. The lack of tool support poses a significant barrier to the adoption of this technology.

Certifying compilation is a technology designed to increase the reliability of software and to enable efficient secure mobile code. A *certifying compiler* is one that compiles a program along with a proof that the program satisfies some property. The output of such a compiler is an object file and a certificate such that a *verifier* can, when given both the certificate and the object file, check that the object file satisfies the given property.

The most commonly checked property is type safety. Type safety ensures that programs only access memory they have allocated, only jump to well-typed code, and respect interfaces. These guarantees allow distrustful hosts to download and safely run programs from unknown sources.

Typed Assembly Language (TAL) [MWCG99] and Proof-Carrying Code (PCC) [Nec97] are two infrastructures for encoding certificates. TAL supports a special purpose higher-order logic that allows type-safety proofs to be concisely expressed for a wide range of programming language constructs. PCC encodes proofs of arbitrary properties in first-order logic. This approach has greater flexibility but the limitation to first-order logic makes certain properties hard to express (for example, properties relating to run-time code generation).

Certifying compilation helps increase the reliability of compilers and the code they generate. In our experience, most compiler bugs cause the output of the compiler to be rejected by the verifier. By catching bugs at compile time and pinpointing their location, the certifying compilation infrastructure can improve the quality of the compiler. Also, programmers may verify the output on their individual programs and thereby gain increased confidence in the correctness of the compiler.

For some optimizations, types can be used to yield improved results [TMC+96]. However, in other cases certifying compilation may effectively limit optimization. Optimizations, such as many array optimizations, incur a greater proof obligation for the compiler writer – the compiler must produce a certificate proving that the optimization is type preserving. If producing this certificate is too difficult, the compiler writer will most likely choose to generate sub-optimal code.

The combination. By combining certifying compilation and run-time code generation, we hope to gain from both of their strengths while ameliorating their limitations. RTCG can be used to improve the performance of certified programs, and certifying compilation can be used to increase the reliability of programs using RTCG.

Luke Hornof and Trevor Jim presented the first certifying compilation system supporting RTCG [HJ99]. Their system consisted of Cyclone[1], a type-safe source

[1] A new version of Cyclone is being developed that addresses some of the weak-

language based on C supporting RTCG; TAL/T, typed assembly language extended with templates; and a non-optimizing compiler from Cyclone to TAL/T. This thesis rounds out that work with three important contributions:

1. An optimizing and certifying compiler from Cyclone to TAL/T.

   To enjoy the full benefits of RTCG it is necessary to generate efficient code. The original Cyclone compiler performed almost no optimizations. Although RTCG gave good relative speedups, the absolute performance of the compiler-generated code was poor.

   We develop a framework for optimizing programs supporting RTCG, and show how to adapt existing control-flow based optimizations to this setting.

2. A formal proof that TAL/T is sound.

   Although the verifier for TAL/T had been implemented prior to our work, it had never been proven sound. Therefore, we could not state with confidence that programs passing the verifier were type safe. In the process of developing our own formalism and proof, we discovered a serious flaw that required correcting the implementation of the verifier.

   Our proof builds on the soundness proof for TAL. We add templates and operations to manipulate templates to the core language. RTCG requires two significant changes: 1) support for values of changing type, and 2) distinguishing relative and absolute jumps.

3. An experimental evaluation of our system.

   Because we are using Cyclone, we had to implement each of the test programs we used. These included a set of numeric kernels, a raytracer and, a simulator. We chose these programs because these categories had all been shown to be good candidates for RTCG in the literature.

   For the numeric kernels we achieved speedups comparable to other systems using similar technologies. For the raytracer, our evaluation showed no performance improvement due to RTCG. For the simulator, we discovered a significant improvement.

   More interesting than these specific results are the larger lessons we learned. Previous experiments in the literature have not presented detailed analyses. By careful examination, we discover that some key factors determine the effectiveness of RTCG:

   (a) Amdahl's law limits end-to-end performance for large realistic applications.

      This folklore lesson is borne out by our experience and well worth restating. Micro-benchmarks tend to overstate gains because the whole

nesses in the original design.

application benefits from RTCG. In larger applications only a fraction of the code may see any improvement.

(b) Instruction cache misses may negate gains due to RTCG.

For small benchmarks, generated code tends to be small and will therefore not severely affect the instruction cache. Our experience suggests that in large applications, dynamic code generation must be carefully controlled to maintain good instruction cache performance.

Therefore, it is essential that systems for RTCG generate dynamic code that is both fast and compact.

(c) Alternatives to RTCG must be carefully considered.

In many cases using a different algorithm may yield the same benefits as RTCG. In those cases, it is important to weigh the complexity of the more efficient algorithm against the complexity of the code annotated for RTCG.

Many instances of RTCG rely on exploiting special values or eliminating computations that are performed repetitively. In those cases data specialization and algorithmic changes are, in our experience, often viable alternatives.

These results are just the first step in achieving a better understanding of RTCG. We believe that even more applications should be tried to clarify the class of applications for which RTCG is truly effective.

This thesis presents these three contributions starting with the experimental results and ending with the formal results. We begin by describing related work and the Cyclone language. Especially interesting in this section is the discussion of how to use Cyclone's RTCG constructs effectively. Then we continue with three chapters presenting our experimental evaluation of Cyclone using micro-benchmarks, a raytracer, and a simulator respectively. After presenting the experimental results, we describe how Cyclone can be compiled and optimized. Finally, we end by describing the formalism we use to prove TAL/T sound. Once TAL/T is formally described the proof is largely standard. It is therefore relegated to Appendix C.

Chapter 2

Background

Machines based on a Von Neumann architecture do not conceptually distinguish between code and data. The machine's state contains a field, the program counter, that points to a memory cell. The bits in that cell are interpreted as an instruction which when executed, among other possible effects, causes the program counter to change. Because of this uniform representation, a program can easily fill memory with data and then make the program counter point to this new data. The process of generating data which is then executed is run-time code generation (RTCG). RTCG is commonly used in operating systems to launch applications; the application data is read into memory and then executed.

Conventional programming languages create a distinction between data and code for the programmer's benefit. The program code is completely specified before the program is run. The code can then manipulate, create, and use data; but data cannot be converted into code. This restriction simplifies reasoning about program execution and does not (in a Turing-complete language) limit expressive power. (Turing-complete languages can create data structures representing themselves and simulate their own behavior using those data structures through interpretation. These encodings do not require RTCG.)

RTCG is, however, useful for improving performance. Dynamically generated code is created in a rich context containing run-time values that are not available at compile time. The benefits of RTCG can often be attributed to eliminating interpretive overhead. For example, an operating system could contain an interpreter for the hardware to which it passed the executable as an input. By using RTCG rather than an interpreter, operating systems achieve better performance.

Because of its potential for performance improvement, RTCG has been studied extensively. This thesis demonstrates how one can use certifying compilation to provide increased assurance for RTCG. This chapter describes existing RTCG and certifying compilation systems.

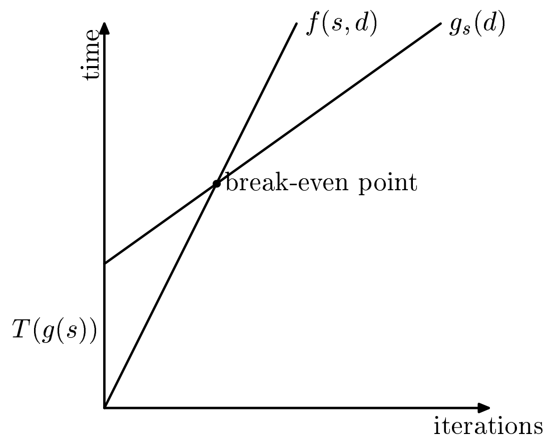


Figure 2.1: The analytical benefits of RTCG.

2.1 The Benefits of RTCG

A popular explanation for the benefits of RTCG is the following [KEH91]: RTCG is run-time partial evaluation. It embeds run-time values into generated code the same way a traditional partial evaluator embeds values at compile time. For this discussion, assume we are given a function $f(s, d)$ and wish to specialize it on its first argument (traditionally called the static argument although the name is misleading in this context). To specialize f we construct a code-generating function g (typically called the *generating extension* of f) such that $g(s)(d) = f(s, d)$ for all values of s and d. The generating extension g is like a partial application of f except that the intended semantics is for g to generate new code. (The Fabius compiler in fact implemented partial application in ML via RTCG.)

We expect performance gains because g can exploit properties of the value s when generating the code for $g(s)$. Let $T(e)$ represent the time to evaluate the expression e. For one run of the overall program, RTCG will prove beneficial for a particular value s if the following inequality holds:

$$\sum_{i=1}^{n_s} T(f(s, d_i)) > T(g(s)) + \sum_{i=1}^{n_s} T(g_s(d_i))$$

In this equation, n_s is the number of times f is called with first argument s, d_i is the value of the second argument on the $i^t h$ call, , and g_s denotes the function produced by executing $g(s)$. We assume that for a given value of s, $g(s)$ is called once only. The inclusion of the term $T(g(s))$ distinguishes RTCG from ordinary partial evaluation and motivates the development of rapid compilation techniques.

This inequality is usually simplified by assuming that f's running time is independent of the value d, in which case the inequality can be rewritten as:

$$n_s T(f(s, d)) > T(g(s)) + n_s T(g_s(d))$$

Both sides are linear equations. Figure 2.1 plots them with n_s on the x-axis and time on the y-axis. The value of n_s at which the two sides of the inequality are equal is called the *break-even point*. The ratio $T(f(s,d))/T(g_s(d))$ is called the *asymptotic speedup*. These two values are often used to measure the performance benefit of RTCG. If the expected value of n_s is below the break-even point than the optimization will not be profitable. The asymptotic speedup provides an upper bound on the speedup.

The following examples demonstrate the kinds of applications in which RTCG might be profitable:

- Interpreters. RTCG can be used to produce a compiled version of the input program.
- Vector and matrix operations. In some applications an operation like dot product or matrix-vector multiply is applied over and over to the same vector or matrix. In these cases RTCG can be used to eliminate multiplications by zero and one, as well as exploiting the structure of the fixed argument.
- Pattern-matching. Patterns are often applied to multiple input sets making them good candidates for RTCG. RTCG can be used to generate special purpose code for matching a particular pattern.
- Encryption. RTCG can be used to create a function embedding a decryption key.

A good candidate for RTCG must have the following properties:

- The overhead eliminated by RTCG must account for a substantial portion of the execution time.
- The inputs to the function must remain invariant over enough calls to recoup the cost of code generation.
- The domain of specialization must be large enough that generating one version of f for each possible value of s at compile time is impractical.

This analysis is helpful for understanding the performance tradeoffs of RTCG but it ignores a number of subtle issues. First, against which f should one compare the generating extension g? For example consider specializing dot product. Suppose g exploits zeroes in its input to eliminate multiplications and additions. Should we compare the performance of g_s to a sparse implementation (using linked lists) or a dense implementation (using arrays) of dot product. The sparse implementation may be competitive with g_s for sparse vectors but will certainly be much worse than g_s given a dense vector. The dense implementation will do worse than g_s on sparse vectors but is likely to be competitive for dense vectors. The appropriate comparison depends on the characteristics of the application in question. We

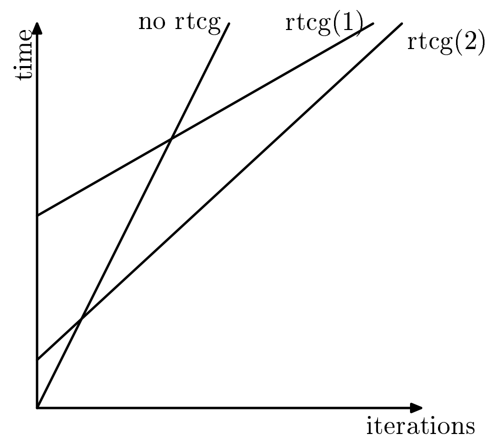


Figure 2.2: Analytical comparison of different techniques for RTCG

should compare to the most appropriate implementation of f for that application. In larger applications (unlike our simple example) this choice may be controversial.

The above analysis also ignores important characteristics of hardware such as caches, and memory limits. Modern architectures use caches to hide the latency of memory accesses. Dynamically generating code increases pressure on the instruction cache by increasing the size of the active code region. Once this active region no longer fits in the cache the program will typically become dramatically slower eliminating any of the benefits gained by RTCG. In cases where very much code is generated the application may exhaust physical memory and begin paging; almost certainly obliterating any performance improvement. Therefore great care is needed when using RTCG to control the amount of generated code.

2.2 Techniques for RTCG

Techniques for RTCG differ in the time they take to generate code, and the quality of the code they generate. If we plot two techniques for RTCG on the same plot as shown in Figure 2.2 these differences are easily seen. The y-intercept is the code-generation time and the slope is the time needed to execute one function call. Based on this plot, research in RTCG has focused on finding techniques for minimizing the code-generation time without sacrificing code quality. We classify some of the popular techniques for performing RTCG.

A straightforward implementation of RTCG has $g(s)$ construct a textual representation of the code it wishes to generate, invoke a compiler on this representation to produce a dynamically loadable file, and dynamically load the file. Given an optimizing compiler, this technique yields well-optimized code and is straightforward to implement.

The above scheme can be improved by (1) eliminating the textual representation and instead using directly the compiler's internal representation, and (2) eliminating the need for dynamic loading. In this *intermediate representation* approach the program includes the compiler. When $g(s)$ generates code, it constructs the code directly in low-level data structures internal to the compiler saving the cost of constructing and deconstructing a textual representation. It then passes this representation to the compiler which returns a pointer to the generated function; thus saving the need for dynamically linking the result. This approach is taken by many Scheme and Lisp compilers. To further reduce code-generation times the compiler may use rapid optimization techniques [PEK97].

Because the set of functions that will dynamically generate code is typically known at compile time, we can generate a compiler specialized to each function we intend to code generate. This specialized compiler is called a *generating extension*. When constructing the generating extension, the compiler can take advantage of the function's structure to perform some optimizations at compile time, and delay others until run time. The complexity of producing a generating extension has limited the optimizations they perform at run time to local optimizations.

Existing systems relying on generating extensions use a particularly stylized form based on *templates*. A template is simply a contiguous block of machine code. A template-based generating extension copies templates together and replaces values in the resulting code to correct jump targets, and replaces operands with appropriate values. Values that are replaced are called *holes*. (Filling holes is similar to the back-patching performed by a traditional loader.) In a pure template-based scheme the generating extension controls only the order and number of the templates it copies, and the values it fills holes with. The simplicity of the key operations, copying and filling, means that code generation is very fast, but the opportunities for dynamic optimization are also limited. Some of these systems also augment code generation with a post-pass that performs local optimizations to overcome this limitation.

Generally, longer code-generation times permit greater optimization. For specific applications this relationship need not hold. For a simple function like dot product, a template-based approach may yield optimal code. Performing additional analyses or optimization passes would only increase code-generation time. At the other extreme, when applying RTCG to an interpreter for a rich language, achieving a speedup may require whole function optimizations of the generated code (such as register allocation).

In this section I have categorized various broad approaches to RTCG. These categories are fuzzy because the different approaches are in a certain sense equivalant. A program could contain one template for each instruction and generate arbitrary code from these. A generating extension could just call a general purpose compiler instead of being truly specialized to a particular function. A compiler based on intermediate representations need not perform any optimizations and could in that case be very fast. Still these four general approaches; dynamic linking, intermediate representations, generating extensions, and templates; represent categories that capture the general approach of most implementations.

2.3 User Interface

Until now, this chapter has focused on the mechanics of generating code. This section describe some of the ways existing systems allow programmers to specify where and how to apply RTCG.

Just-in-time compilers [BCF+99, Mic99] and dynamic optimization systems [Kis99, Kis97] profile the execution of a program on-the-fly and generate optimized code for frequently executed paths. These systems require no programmer annotations and can achieve significant performance gains. However, because there is no programmer control, complex domain specific optimizations are unlikely to be discovered or applied.

The next level of programmer control asks programmers to annotate the function parameters they wish to drive dynamic code generation. A binding time analysis [JGS93] is then used to flow this information from the parameters through the body of the function. The result of the analysis is a staged function where computations that can be performed at compile time are marked as static, and the rest are marked as dynamic.

Even with this level of control, the compiler may miss certain optimizations. For example, a binding-time analysis may have determined that in the expression $x * y$ the value of x (but not y) will be known at code-generation time. If x is 0 than $x * y$ can be simplified to 0 independent of y. Certain code-generation strategies (such as a template-based approach) may not automatically apply this optimization. They will generate the multiplication whatever value x takes. If eliminating this multiplication is important the programmer may have to rewrite the expression $x * y$ as `if x=0 then x*y else 0`. The technique of inserting additional tests to make optimizations apparent applies so commonly in partial evaluation that it is simply called *the trick*. Knowing where to apply *the trick* requires a deep understanding of the compilation model.

Instead of applying *the trick*, programmers can explicitly specify where and how to apply RTCG; and even what optimizations to apply. Although more intrusive, this approach allows very fine-grained control. Our system takes this approach. We have not found it onerous.

Another dimension of the user's interaction with RTCG is the choice of host language. RTCG is usually presented as an extension to an existing programming language. When hosted within a high-level language, like ML, the user gains the convenience of high-level specification but loses the ability to predict easily the code that will be generated. For a system using heavyweight RTCG, the programmer may feel confident that the dynamically-generated code is well optimized. For a lightweight approach, this choice is problematic because so many factors determine the profitability of RTCG. When hosted within a low-level language, like C, the user must potentially deal with more low-level details but gains a good model of how code is generated and can more easily understand the performance implications of her choices. C in conjunction with a binding-time analysis is a popular choice because it allows many applications to be rapidly ported.

The user's experience is also affected by the development environment. Writing programs with RTCG is hard. Consequently tools that help catch bugs are invaluable. Currently there are no debuggers or profilers suitable for debugging or profiling programs using RTCG.

Fortunately, type systems capable of ensuring that programs containing RTCG are well staged and well typed are well understood via a Curry-Howard isomorphism with modal logics [Dav96, WLPD98]. In these type systems, the modal operator on a type ($\Box\tau$) indicates a suspended code fragment. These code fragments may be combined. Dynamic code is generated via a builtin operator (usually called `eval` or `run`) that takes a value of type $\Box\tau$ and produces a value of type τ.

Type systems have been proposed based on the modal logic $S4$ and linear temporal logic. The former only allows the manipulation of closed code fragments but admits `eval`. The latter allows open code fragments to be manipulated but does not admit `eval`. Recent work has combined the two [Tah00].

These formal systems are based on the λ-calculus. In that model, RTCG can be best understood as allowing the programmer to specify a non-standard order of evaluation. The key property that these modal type systems enforce is that variables are used in the "right stage." For example, if a variable x is bound within a suspended code fragment then the function "generating the code" cannot access x.

MetaML [TS97] and PML [WLP98] are examples of languages supporting type-safe RTCG based on these formalisms. Although Cyclone is safe too; it does not introduce new code types and is not directly related to these modal logics.

2.4 Related Work: RTCG

Now we present related systems for RTCG using the terminology developed in the earlier sections.

Interpreted languages often support a mechanism for interpreting an arbitrary dynamically constructed string. The most well-known of these is the `eval`, `quote`, and `backquote` mechanism in Lisp and Scheme designed to simplify such construction. This feature is both powerful and, in an interpreter, easy to implement; simply pass the string to the interpreter.

When performance matters most of these interpreted languages also have compilers. These compilers usually rely on heavyweight RTCG; they include the compiler in the executable. This technique is widely used so we cite only a little relevant work [KKR+86, Ste78].

Tempo [NHCL96, CHN+96, CD93, CN96] is a partial evaluator for C that can be configured for RTCG. Based on user annotations indicating which function arguments should be treated as invariant, Tempo performs a sophisticated binding time analysis to stage its input. The input is then compiled into a C program that performs RTCG. Through a clever compilation technique Tempo is able to leverage an existing compiler to perform template-based RTCG. The dynamically generated parts of the program are translated into special-harness C code that is compiled by the C compiler at compile time. The templates (sequences of machine code instructions) are lifted directly from the resulting object file and inserted as data into the generating program.

Our language, Cyclone, was originally designed as a back end for Tempo. The language constructs are sufficient for this purpose; although we have not made this linkage.

DyC [APC+96, GPM+99, GMP+97, GMP+00] is an extension of the Multiflow [LFK+93] optimizing compiler with RTCG. DyC, too, relies on a binding time analysis to stage the program. Unlike Tempo, DyC's analysis permits polyvariant specialization – a given program point may be specialized in different ways depending on the context. Polyvariant specialization permits more optimization at the cost of code duplication. DyC performs most analyses and optimizations before staging, and tracks the user's annotations through optimization.

DyC uses an extended template-based approach. Although templates are generated at compile time some local optimizations are performed at run time. These optimizations include eliminating dead assignments and strength reduction. These run-time optimizations may change the size of the resulting code and therefore require that DyC dynamically rewrite all jump targets.

'C [PHEK99, Pol99, EHK96, PEK97, PEK96] is an extension of C with explicit support for RTCG. The 'C compiler, tcc, does not perform any analysis to determine where RTCG should be applied. The programmer must write his programs using RTCG constructs explicitly. tcc can be configured to compile 'C programs with either a lightweight or a heavyweight backend. The heavyweight backend takes the compiler's intermediate representation and applies all the optimizations used by the compiler at compile time. The lightweight backend uses the same representation but applies a series of fast linear optimizations such as linear scan register allocation [PS99b], and linear data analyses. These analyses do not achieve the same precision as traditional optimizations but can be computed more efficiently.

MetaSML [TS97, Tah00, TBS98, Tah99b] adds explicit RTCG constructs to ML. The primary research aim of the MetaSML compiler has been to develop better type systems for RTCG. Because they have not focused on lightweight RTCG, the compiler simply calls itself at run time to generate code.

Fabius [LL94, LL96, LL98] was one of the first RTCG systems. It used RTCG to optimize closures in ML based on user annotations. Fabius compiled down to generating extensions that performed local optimizations while generating code.

Unfortunately, Fabius never supported garbage collection and therefore could not run large ML programs. For the small benchmarks it did run, it achieved spectacular results.

PML [WLP98] is an extension of ML supporting RTCG. It was developed to explore research issues in type sytems. PML compiles to an extended Categorical Abstract Machine (CAM). The published results for PML show significant improvements in the number of CAM reduction steps through using RTCG. As the descendant of Fabius, PML plans to use generating extensions accepting a compilation context, therefore permitting significant optimization at run time.

2.5 Certifying Compilation

In a language like 'C programmers may assemble almost any dynamic code. The type system will not check any properties of this generated code. Without tool support, discovering and correcting bugs in the presence of RTCG is extremely difficult. Tempo and DyC ameliorate this problem by ensuring that annotated programs behave like their unannotated versions. Therefore programmers can debug the unannotated version relying on traditional means before adding the annotations.

An even stronger guarantee is provided by MetaML and Fabius. In these languages, the type system guarantees that well-typed programs will not crash. However, these guarantees rely crucially on the compiler being correctly implemented.

To be sure of this type-safety guarantee, we would like to prove the correctness of our implementation. This goal is at present unattainable. A weaker approach is to check that the output of the compiler is type safe. This goal can be accomplished via certifying compilation.

Certifying compilation is a framework for producing a certificate that can be used to check the output of a compiler with respect to a particular property. A certificate together with its associated machine code constitutes a proof of the property in question. Although this framework applies to any property, existing certifying compilers only establish one property: type safety.

A certifying compiler compiles a type-safe language into certified output by compiling the proof of type-safety provided by the user (in the form of types) along with the code. Thus each transformation in the compiler must be accompanied by a type translation. Work on type-directed compilation which preceded certifying compilation has shown that types are useful both for debugging compilers, and directing optimizations [TMC+96].

There are several implementations of certifying compilation:

1. The Proof-Carrying Code (PCC) system [Nec97, NL98a, NL96] represents proofs directly in first-order logic. First-order logic can be used to prove many different program properties including type safety and limits on resource

bounds. The certifying compiler Touchstone is used to compile a type-safe C-like language and generate x86 object code along with a proof of type-safety [NL98b].

2. Typed Assembly Language (TAL) [MWCG99, MCG+99] is assembly language annotated with typing annotations. The TAL verifier uses these annotations to check type safety. TAL supports explicit initialization of memory [SWM00], stacks [MCGW98], modules [GM99], and array-bounds check elimination through singleton types [XP98, XP99]. Like other certifying compilation systems TAL relies on a garbage collector, and a built-in allocation primitive.

   On paper, TAL has been proven sufficiently expressive to compile the polymorphic λ-calculus (System F [Gir71, Rey74]). In practice, an optimizing and certifying compiler from a rich C-like language generating TAL for the Intel x86 architecture has been constructed [GM00].

   TAL's types in essence represent a special purpose higher-order logic useful for proving type-safety. This logic is tailored to make proofs of type-safety concise (relative to the proofs for PCC for example), and to permit common language constructs to be easily checked. Over time, the properties encompassed by this logic have grown stronger. For example, research has been done on extending TAL to support resource bounds [CW00].

   This thesis extends TAL with support for RTCG.

3. Foundational Proof-Carrying Code [AF00, App01] aims to provide proofs built from first principles. TAL and PCC were proven sound with respect to abstract machine models. This work aims to prove correctness based on a formal model of the hardware. By using higher-order logic, automated proof checkers and theorem provers they have been able to handle the complexity of this task.

4. Efficient Code Certification (ECC) [Koz98, Koz99] focuses on producing small certificates and easy integration with existing compilers. By encoding compiler invariants in the checker, certificates can be made smaller and modifications to the compiler to produce the certificates minimized. The drawback of this approach is that it is brittle with respect to changes in the compiler's code generation strategy.

Chapter 3

Cyclone: A Source Language for RTCG

Cyclone is a type-safe C-based language that allows programmers to express RTCG directly and safely. We designed Cyclone to meet particular criteria:

- Cyclone is a full-fledged language.

  To perform meaningful performance studies or make credible claims that the technology scales, it must be possible to port significant applications into Cyclone. We have ported an instruction set simulator, and a raytracer into Cyclone. In addition, the optimization techniques developed in this thesis were tested in a compiler for Cyclone written in Cyclone.

- Compiling Cyclone into TAL/T is straightforward.

  To explore certified RTCG we planned to compile the source language into an augmented TAL with support for RTCG. Compiling existing languages into TAL poses a number of challenges. To compile Java into TAL, a number of technical obstacles (like object encodings) would have to be overcome [Gle99]. Similarly, compiling ML into TAL requires handling ML's module system. The theoretical underpinnings of ML's module system are the subject of ongoing research. Compiling other unsafe languages like C or C++ into a type-safe language is not yet practically feasible.

- Cyclone provides the programmer with low-level semantics.

  Because RTCG is about optimizing performance, we felt that it was important to have a simple model of code generation. Like C, most Cyclone constructs have clear costs. For example, builtin constructs do not implicitly allocate memory. Primitive operations correspond directly to machine instructions. Function local variables are explicitly declared and, as one would expect, register allocated.

  These choices all the programmer to easily predict the code that will be generated and perform many optimizations manually. As we will see later,

```
// Search an array of pairs for the pair with first component equal to x.
// Raise an exception if no such element.
// The type *(int,string) denotes a pair of an integer and a string.
*(int,string) search(*(int,string) a[], int x) {

  _ n = size(a);
  for(int i = 0; i < n; i++) { // size is buit-in for arrays.
    if(a[i].1 == x) return a[i]; // .1 extracts the first element of a tuple.
  }
  raise Not_Found; // raise an exception Not_Found on failure.
}
```

Figure 3.1: First example Cyclone program.

the effect of RTCG is sensitive to many factors; making this level of control useful.

Based on these considerations we designed Cyclone: a type-safe C-like language supporting parametric polymorphism, exceptions, algebraic datatypes, and RTCG. Because all of our benchmarks are written in Cyclone and all of our examples are given in Cyclone it is important to understand the key features of the language. The remainder of this chapter explains the aspects of the Cyclone language and our compiler that are relevant for understanding the performance results presented in Chapter 4, Chapter 5, and Chapter 6. This chapter is not intended to serve as a language manual.[1]

3.1 The Base Language

Cyclone is a type-safe C-like language supporting parametric polymorphism, exceptions, algebraic datatypes, and RTCG. The core statement and expression forms are analagous to C [KR88]. Figure 3.1 shows a simple function that searches an array of pairs and returns the first matching element. Many aspects of this function are identical to a C version. There are some important differences:

- The argument `*(int,string) a[]` is an array argument to the function `search`. The size of any array can be determined using the builtin routine `size(a)`. Arrays in Cyclone carry their size with them for the purpose of checking array bounds.
- The compilerautomatically infers omitted types when possible. To omit a type the programmer writes an underscore.

[1] Based on our experiences with this version of Cyclone we are designing a new version that is more powerful and expressive.

- The type `*(int,string)` denotes a tuple consisting of an integer and a string. If the variable `x` had this type, then `x.1` would return the integer and `x.2` would return the string.

  Tuples can contain one or more elements. The type `*(int)` is a pointer to tuple containing only one integer. Tuples are useful for returning multiple arguments without having to declare a new structure type.

- Cyclone supports exceptions as demonstrated by the `raise Not_Found` expression.

There are many more differences between Cyclone and C. For understanding our examples it suffices to understand C, the above differences, and the RTCG constructs described later in this chapter. However, to appreciate the results of our experiments it is useful to understand some of the design decisions that have performance implications.

3.1.1 Primitive Types

All the usual C primitive types; `char`, `short`, `int`, `float`, and `double`; are supported. And all the precedence rules of C are obeyed. However, when stored in memory, values of these types consume at least 4 bytes. In C, a `char` stored in a `struct` may only occupy one byte if no padding is necessary for alignment reasons. Cyclone does not forbid such a representation but makes it practically difficult to use one.

Parametric polymorphism interacts poorly with such special purpose data representations. Consider a function taking a polymorphic array as an argument. That function must be able to access the elements of the array without knowing the element's type. This implies that the size of the elements cannot depend on the type. Therefore, an array of characters (`char x[]`) must naively be represented as an array of 4-byte values.

Better representations are possible. One solution is to perform an analysis to determine when a more compact representation can be safely used [Mor95, Ler92, TMC$^+$96]. Another solution is to duplicate code for polymorphic functions for each case (as is done in C++). A final solution is to forbid instantiation of polymorphic type variables with types representing values of unusual size. The latter solution is used for doubles in Cyclone. A `double` by necessity occupies 8 bytes, and therefore Cyclone forbids any type variable from being instantiated with a `double`.

3.1.2 Unions (or Algebraic Datatypes)

The `union` type allows programmers to declare a type for a value that may have one of many forms. These are especially convenient for declaring abstract syntax. Figure 3.2 shows how a union type might be used to represent the λ-calculus, and also use the resulting term. The type `*(exp,exp)` denotes a pair containing two

```
// Declaration of a union type exp.
union exp {
 string Var;
 *(exp,exp) App;
 *(string,exp) Lam;
}

// Example code showing how to use values of union type.
switch x {
 case Var(v): ; // v has type string.
 case App(v): ; // v has type *(exp,exp)
 case Lam(v): ; // v has type *(string,exp)
}
```

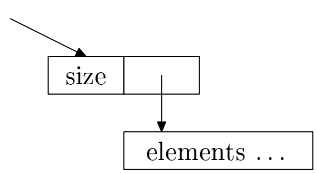
Figure 3.2: Example use of Cyclone the `union` type.

values of type `exp`. The `switch` statement allows the programmer to deconstruct the value and extract its componenets.

Unlike C unions, Cyclone unions are checked. In the implementation, this checking is achieved by associating an integer with each case. The switch statement determines the type of the carried value based on this integer. This checking ensures that programs do not treat values of union type incorrectly, but also incurs some overhead.

3.1.3 Arrays

Cyclone arrays are like C arrays except that array accesses must be checked to ensure that the index is within the array's bounds. In the implementation, this checking is achieved by recording the size of the array with the array elements. To simplify communicating with C code, our implementation uses the following representation for arrays.



3.2 Adding RTCG

Cyclone uses just four new RTCG constructs: `codegen`, `cut`, `splice`, and `fill`. These constructs are best conceptualized as manipulating "strings" to construct the text of a new function (though of course this is not how things are done). For

```
int dot_gen(int u[])(int []) {
  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;
  });
}
```

Figure 3.3: An RTCG version of dot product.

example, the function `dot_gen` in Figure 3.3 takes a vector `u` and returns a function specialized for taking dot products of that particular vector `u`. (The outer boxes are used to highlight code fragments that are dynamically assembled to produce the specialized function. The inner boxes highlight holes that are filled with values during run-time code generation.)

The expression `codegen` $(int(dot(int\, v[\,])\ldots)$ indicates RTCG of a function. The generated function will be composed of copies of code fragments (templates) from within the `codegen` expression. We have highlighted the templates by surrounding them with boxes.

The `codegen` expression takes an argument that is a function syntactically but it is not executable. It contains templates, the first of which indicates the arguments and return type for the generated function. In our example, this template also includes the declaration of `result`. The code in the template is not executed during the invocation of `dot_gen`; instead, it is copied as the beginning of the generated function. The `cut` statement marks the end of the first template. The code within the `cut` is executed during the invocation of `dot_gen`. It iterates over the array `u`, executing a `splice` statement for each element. The `splice` statement marks the beginning of a template; its argument is appended to the end of the function being generated.

In our example, this argument contains `fill` expressions. These expressions mark *holes* in the template that are filled with values calculated by `dot_gen`; we have highlighted these holes with nested boxes. Finally, after the loop is finished executing, a copy of a third template ends the generated function. The result of the `codegen` is a pointer to the generated function.

Invoking `dot_gen` on an array with elements {`12,24,32`} would generate a function like the one shown in Figure 3.4. In this example, RTCG allows us to eliminate the looping overhead, fold constants into the instructions, and eliminate

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

Figure 3.4: The function conceptually produced by `dot_gen({12,24,32})`.

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

Figure 3.5: A more aggressive version of `dot_gen` that specializes for 1 and 0.

array-bounds checks on accesses to `u`. A more aggressive version, like the one shown in Figure 3.5, could generate different code when an element of `u` is zero or one.

These 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.

$$
\begin{array}{c}
\text{Cyclone Input} \\
\Downarrow \\
\boxed{\text{Parse}} \\
\Downarrow \\
\boxed{\text{Type check}} \\
\Downarrow \\
\boxed{\text{Translate into CIR}} \\
\Downarrow \\
\boxed{\text{Optimize}} \\
\Downarrow \\
\boxed{\text{Register Allocate}} \\
\Downarrow \\
\text{TAL/T Output}
\end{array}
$$

Figure 3.6: Stages of the Cyclone compiler.

$$
\begin{array}{ll}
t_1: & result := 0;\ \underline{\texttt{jmp}} \circ \\
t_2: & x := [h_1];\ i := \underline{[h_2]};\ tmp := x * v[i];\ result := result + tmp;\ \underline{\texttt{jmp}} \circ \\
t_3: & \underline{\texttt{retn}}\ result;
\end{array}
$$

Figure 3.7: Pseudo-code for the three templates used by `dot_gen` (without prologue or epilogue code).

3.3 The Cyclone Compiler

The Cyclone compiler uses a template-based approach to rapidly generate code at run time. The idea is to optimize the dynamically generated functions at compile time; thereby making dynamic code generation fast.

Figure 3.6 shows the stages of our compiler. After parsing and type-checking the code, the compiler translates Cyclone into an intermediate representation (CIR) suitable for optimization. After performing standard optimizations, the compiler register allocates the code (including the dynamically generated functions) to produce TAL/T code. Each dynamic function is represented by a set of machine-code templates.

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

Figure 3.7 shows roughly how these templates might look in our compilers intermediate representation. The labels h_1 and h_2 name the holes in template t_2.

Figure 3.8 shows how the body of `dot_gen` might use these templates to gener-

b_1 : $r :=$ <u>`start`</u> dot; $p_1 :=$ <u>`copy`</u> t_1 <u>`into`</u> r; $i := 0$; <u>`jmp`</u> $\circ$
b_2 : <u>`jnz`</u> $i < size(u)\,?\, b_4 : \circ$
b_3 : $p_2 :=$ <u>`copy`</u> t_2 <u>`into`</u> r; <u>`fill`</u> $p_2.[h_1]$ <u>`with`</u> $u[i]$; <u>`fill`</u> $p_2.[h_2]$ <u>`with`</u> i;
$i := i + 1$; <u>`jmp`</u> b_2;
b_4 : $p_3 :=$ <u>`copy`</u> t_3 <u>`into`</u> r; $f :=$ <u>`end`</u> r; <u>`retn`</u> f;

Figure 3.8: Pseudo-code for the body of `dot_gen`.

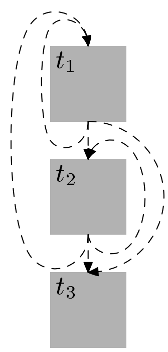


Figure 3.9: All ways control might flow between `dot`'s templates. No edges flow out of t_3 because it ends in a `return` statement. The edges (t_2, t_1) and (t_1, t_1) are impossible only because of the way `dot_gen` assembles the templates.

ate code. The labels b_i name the blocks in `dot_gen`. The $r :=$ <u>`start`</u> *dot* instruction allocates a new code buffer called r. The $p_1 :=$ <u>`copy`</u> t_1 <u>`into`</u> r instruction appends the template t_1 to the buffer r and returns a pointer to the copy in p_1. The instruction <u>`jmp`</u> $\circ$ falls through to the next block in the sequence. The fill instructions are used to fill the holes in the copy of the template t_2 appended to the code buffer on that iteration of the loop. Finally on exiting the loop in block b_4, `dot_gen` appends t_3 to the buffer, stops code generation with the $f :=$ <u>`end`</u> r instruction (this instruction places a pointer to the newly generated code in f), and returns f.

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

For example, Figure 3.9 depicts *all* ways control might flow between `dot`'s templates. By analyzing `dot_gen`, we discover that every specialized function it produces will have a restricted form: It will start with one copy of the first template, followed by some number of copies of the second template (appropriately specialized with constants), and then one copy of the third. In other words, the edges (t_2, t_1) and (t_1, t_1) in Figure 3.9 could not be followed. Such an analysis would allow us to compute more accurate live ranges for the function's variables.

Our compiler uses the analysis framework we develop to compile the CIR templates into machine code templates in TAL/T. These machine code templates are register allocated and optimized just like all the other code produced by the compiler.

Figure 3.10 shows the TAL/T code for the templates produced by compiling `dot_gen` with our compiler. In this example we have not included any of the typing information, and have renamed labels to be consistent with our previous examples. TAL/T includes built-in support for manipulating templates via operations similar to those present in CIR.

This code shows that the compiler has successfully allocated all the temporary variables to registers. This code is the best our compiler could produce for any function; dynamically generated or not.

Figure 3.11 shows the body of `dot_gen` in TAL/T as generated by our compiler. This function allocates a code buffer, copies (or dumps) templates into it, and fills holes. These are the only operations in TAL/T that can be used to dynamically generate code. Therefore code generation is very fast.

We have informally described how our compiler generates code and optimizes dynamic functions at compile time. In Chapter 7 and Chapter 8 we describe more formally how to compile Cyclone into CIR, and how CIR may be optimized.

3.4 Programming Techniques

Cyclone was designed for simplicity. The language does not require special types for code and relies on just four RTCG primitives. However, in so doing, it loses some flexibility. In this section we describe programming techniques to overcome some of the drawbacks of the design.

3.4.1 Modularity

The most glaring deficiency of the language is the lack of modularity. Because Cyclone does not have types for code fragments, code cannot be passed to, or returned from functions. Consequently all the code fragments used to generate a dynamic function must reside syntactically within its `codegen`.

This was a concious choice in the design of Cyclone. It enables a simpler type system. Types for code fragments are complex in a C-like language. We do not believe programmers would be willing to write them down. This choice also permits greater (and easier) optimization, as demonstrated in Chapter 8. Note that programmers can use helper functions to generate code; they simply cannot inline that code into another generated function.

To make this code more readable all typing annotations have been removed, labels have been renamed, some liberties have been taken with the syntax, and comments have been added.

```
TEMPLATE t_1
_dot:
        PUSH    EBX
        PUSH    ESI
        MOV     EAX,DWORD PTR [ESP+12]
        MOV     EDX,DWORD PTR [EAX+0] ; size of v
        MOV     EBX,DWORD PTR [EAX+4] ; pointer to v
        MOV     EAX,0
TEMPLATE END

TEMPLATE t_2
entry$1:
        CGHOLE  ESI,h_1  ; hole for u[i]
        CGHOLE  ECX,h_2  ; hole for i
        CMP     EDX,ECX  ; i < size(v)
        JA      beyond$1
        MOV     EAX,_array_bounds_error
        JMP     EAX
beyond$1:
        MOV     ECX,DWORD PTR [EBX+4*ECX+0]  ; read v[i]
        IMUL    ECX,ESI                      ; i * v[i]
        ADD     ECX,EAX
        MOV     EAX,ECX                  ; Add is commutative?
TEMPLATE END

TEMPLATE t_3
entry$2:
        POP     ESI
        POP     EBX
        RETN
TEMPLATE END
```

Figure 3.10: TAL/T templates generated by our compiler for `dot_gen`.

To make this example readable, typing annotations are not included, labels have been renamed, and comments have been added..

```
        CGSTART              ; Allocate a new code buffer in EAX.
        CGDUMP  EAX,tp$0,ECX,t_1   ; Append t_1 to code buffer.
b$2:
        CMP     EBX,DWORD PTR [ESP+0]
        JGE     b$4
b$3:
        CGDUMP  EAX,tp$1,EDI,t_2
        MOV     ECX,EBX
        CMP     EDX,ECX
        JBE     _array_bounds_error
        MOV     ECX,DWORD PTR [ESI+4*ECX+0]
        CGFILL  EAX,EDI,t_2,h_1,ECX
        CGFILL  EAX,EDI,t_2,h_2,EBX
        ADD     EBX,1
        JMP     b$2
b$4:
        CGDUMP  EAX,tp$2,ECX,t_3
        CGEND   EAX
        ; EAX now contains the generated function.
```

Figure 3.11: Body of `dot_gen` compiled into TAL/T by our compiler.

```
int compile_code(int pc)() {
  return (codegen void execute(int sp) {
    cut {
      while(true) {
        switch parse_instruction(read_mem(pc)) {
        case Add *(offset1,offset2,offset3):
            splice {
              int a1 = read_mem(sp + fill(offset2));
              int a2 = read_mem(sp + fill(offset3));
              write_mem(sp + fill(offset1),a1 + a2);
            }
        // Many cases omitted.
        case Jump(l): // What do we do here?
        }
      pc = pc + 4;
      }
    }
  });
}
```

Figure 3.12: Sample code for a simple interpreter.

3.4.2 Control-flow

The available language constructs also introduce overheads in the compilation of control-flow and recursive data structures. A small example should make the problem clear. Consider a simulator for a processor, and for simplicity assume this processor only has two registers, a program counter (`pc`) and a stack pointer (`sp`). We further assume the existence of functions for parsing instructions from memory, and reading and writing to memory. Figure 3.12 shows pseudo-code for a specialized version of the interpreter's main loop.

When we try to generate code for `Jump` we appear to be stuck. The Cyclone constructs only allow us to use structured control-flow constructs, but we are trying to compile an interpreter supporting completely unstructured control-flow.

This problem can be solved through *trampolining*. Trampolining is used when compiling functional languages into C to limit the growth of the stack in tail-recursive functions. The idea is to have the case for `Jump` return the address of the target destination to a *trampoline* function. The trampoline function looks up the address of the destination to see if it has already generated a function for that address. If it has, it calls that function. Otherwise, it generates code for that target and then calls it. Figure 3.13 shows a simple function implementing trampolining. (Trampolining is used by the RTCG versions of both the raytracer

```
struct state {
  int pc;
  int sp;
}

void trampoline(state s) {
  while(true) {
    int f(state);
    if(generated_code(s.pc)) {
      f = lookup_code(s.pc);
    }
    else {
      f = compile_code(s.pc);
      insert_code(s.pc,f);
    }
    s = f(s.sp);
  }
}

state compile_code(int pc)(int sp) {
  return (codegen state execute(int sp) {
    // ⋮
      case Jump(l): splice { return new state(fill(l),sp); }
        // ⋮
  });
}
```

Figure 3.13: Example Cyclone code for trampolining.

in Chapter 5 and the simulator in Chapter 6.)

For each jump in the simulated program the above code returns from the generated code, performs a lookup, and then makes a function call. Because the destination for the `Jump` is known while generating the code, we can optimize away the lookup by performing it at code-generation time instead (if we are careful). If the destination happens to be the function currently being generated, the lookup will not find the generated code and will enter an infinite loop: generating code for a particular pc, and then trying to find that code before it is done. One solution is to insert a dummy function before generating the code so that the lookup will succeed, and then replace the dummy function when code generation is complete.

3.4.3 Array Accesses

RTCG can be used to eliminate bounds checks for array accesses provided that the index and the array are statically known. Suppose that the array is in variable `x` and the index is variable `i`. It may appear that we can simply write `fill(x[i])` to eliminate the array access. In some cases this may be correct, but if the values in `x` change over time this subsitution fails. In particular, the assignment `fill(x[i]) = 3` will not type check.

We can use a simple type transformation to accomplish our goal. If the type of `x` is `int []`, we rewrite the type to `*(int)[]`. (Recall that `*(int)` is the type of a one-element tuple.) Now, our goal is easily accomplished. We replace `x[i]` by `fill(x[i]).1`. (Recall that `.1` extracts the first element of the tuple.) This translation preserves the semantics of the original array even when on the left-hand size of an assignment and extends to array elements of any type.

3.4.4 Controlling Code Explosion

Sometimes eagerly generating code at every opportunity can result in a code explosion that reduces application performance by swamping the instruction cache. For example, consider optimizing vector-matrix multiply where we assume the matrix is known in advance. If the problem uses a large number of matrices generating a new function for each matrix may be unwise. On the other hand, our optimized function may only take advantage of zeroes and ones in the matrix. If that is the case, two matrices with zeroes and ones in the same places could share the same code. The solution is to compute a signature from a matrix that records where it contains zeroes and ones and use that to generate code. Then instead of generating one function per matrix, we generate one function per signature. This technique is generally applicable. For the data structure in question, compute a signature capturing the aspects that can be optimized and generate one function per signature.

Chapter 4

Micro-benchmarks

Micro-benchmarks are useful tools for understanding the effectiveness of an optimization. Their simple structure allows a precise analysis of the observed behavior. And, they can be tailored to explore cases of interest. On the other hand, results from micro-benchmarks tend to overstate the magnitude of improvements. In large applications, Amdahl's law (alternatively the law of diminishing returns) tends to limit the end-to-end performance improvement.

The six micro-benchmarks we chose are either ported versions of those used to evaluate Tempo [NHCL96] (`cheb`, `csi`, `fft`, `romberg`) or standard algorithms (`dot`, `poly`). The Tempo benchmarks are all mathematical kernels [GNZ95].

- Chebyshev (`cheb`) computes the coefficients of a chebyshev polynomial approximating an arbitrary function: The more coefficients, the better the approximation. We specialize the code on the desired number of coefficients.

- Cubic spline interpolation (`csi`) takes a set of two-dimensional points and computes a spline passing through them. We specialize the code by assuming the x-coordinates are fixed.

- Fast-fourier transform (`fft`) takes a sequence of points whose length is a power of two and computes their transform. The transform is the same length as the input vector. The restriction to powers of two enables a divide-and-conquer strategy. We specialize the code on the length of the input.

- Romberg (`romberg`) performs romberg integration of a one-argument user-supplied function to a desired level of accuracy over a given range. We specialize the code over both the range and the desired precision.

- Dot-product (`dot`) computes the dot product of two vectors. We present numbers for both an integer version (`dot`) and a floating-point version (`dotf`). The code is specialized on the value of one of the vectors. Our input vectors vary in length (and except when explicitly stated otherwise) contain 20% zeroes.

- Polynomial (`poly`) computes the value of a polynomial given an array of coefficients. We also present numbers for an integer version (`poly`) and a floating-point version (`polyf`). The code is specialized on the array of coefficients (which contains no special values).

The full source code for the RTCG and non-RTCG versions of these benchmarks is presented in Appendix A. Figure 4.1 duplicates `poly`'s source code here for easy reference. As in most of these examples, the RTCG and non-RTCG versions are very similar.

We use these micro-benchmarks to evaluate the effectiveness of both the compiler's optimizations, and run-time code generation. There are four versions of each benchmark: 1) a plain Cyclone version without RTCG, 2) a Cyclone version using RTCG, 3) a strightforward port into C, and 4) a port into C that has the same safety guarantee as the Cyclone version. This last safe-C version performs the same array bounds checks, uses the same array representation, and relies on the same garbage collector as Cyclone. The two Cyclone versions can further be compiled either with the optimizing register-allocating compiler described in this thesis, or a non-optimizing stack-based compiler.

Methodology. All experiments were run on a computer running Linux Redhat 7.0 on an Intel Pentium III 750 Mhz processor with 128 MB of main memory, a 16KB first-level instruction cache, 16 KB data cache, and a unified 256 KB L2-cache.

We relied on the processor's time stamp counter to obtain accurate cycle counts. The test harness was setup to call each inner function 10 times and average the resulting cycle counts. At least three timing runs were made for each version. These numbers showed little variance. The best time is reported in all cases.

4.1 Comparison of Cyclone to C

To help determine the quality of the code generated by our compiler we compare its performance against the performance of similar code for the C compiler. Figure 4.2 compares the performance of two C versions to the optimized Cyclone version for each benchmark. The unsafe C version is a straightforward C implementation of the benchmark, whereas the safe C version uses the same data representations, uses the same garbage collector, and performs the same bounds checks as the Cyclone version. Both C versions are compiled with `gcc -O2`. For each benchmark we have chosen a representative input value (the results are similar for other input values). For most of the benchmarks the Cyclone version lies between the two C versions suggesting that our optimizations are competitive with the optimizations performed by `gcc`. To match the best C performance we would have to further eliminate bounds checks, and improve our data representations.

The Cyclone compiler achieves better performance than the safe C version because it is aware of array bounds checks. The Cyclone compiler eagerly puts

No RTCG:

```
int poly(int p[],int x) {
  int len = size(p);
  int x_i = 1;
  int result = 0;
  for(int i=0; i<len; ++i) {
    _ ai = p[i];
    if(ai != 0) result += ai * x_i;
    x_i *= x;
  }
  return result;
}
```

RTCG:

```
int poly_gen(int p[])(int) {
  int len = size(p);
  return codegen (int poly(int x) {
    int x_i = 1;
    int result = 0;
    cut {
      for(int i=0; i<len; ++i) {
        _ ai = p[i];
        if(ai != 0) splice { result += fill(ai) * x_i; };
        splice{ x_i *= x; };
      }
    }
    return result;
  });
}
```

Figure 4.1: Source code for `poly`.

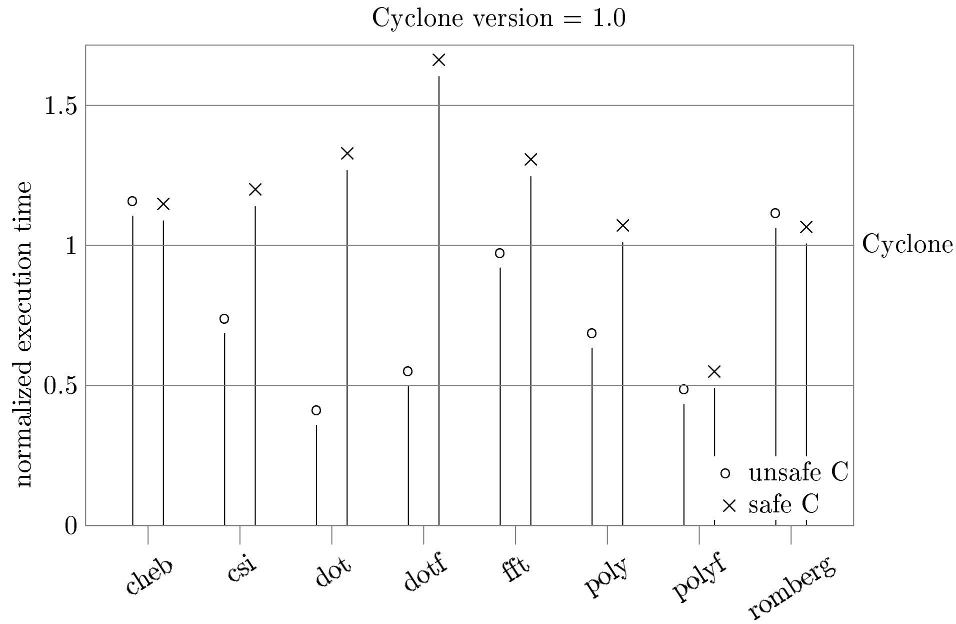


Figure 4.2: Performance of the unsafe C version, the safe C version, and the Cyclone version.

the size and pointer to the array body into new temporaries for each local array variable. This optimization potentially saves two memory operations for each array access, but is only safe because both the size and the pointer to the array body are constant. The C compiler does not know these values are constant and therefore cannot perform this optimization without further analysis. (Note that declaring a variable constant in C does not guarantee that its value will not change and therefore is not sufficient to allow the C compiler to perform this optimization.)

For `polyf`, the Cyclone compiler does much worse than the safe-C version compiled with `gcc` and, for `cheb`, it does much better. Both of these differences are due to compilation strategies for floating-point code. In `cheb`, the Cyclone compiler aggressively uses built-in trigonometric instructions whereas `gcc` uses math library routines. In previous Intel processors, library routines were faster and more accurate than the built-in instructions but that is no longer the case [Cor97]. In `polyf`, the difference is a failing of the Cyclone compiler's heuristics. Backward compatibility has left the Pentium with a floating-point architecture based on a stack model rather than a conventional register architecture. Compiling into this model is a black art. The Cyclone compiler establishes inter-block invariants to simplify the compilation. In `polyf`, the inner loop has only 34 instructions, and of these 12 re-establish the invariant but do no useful work. The C compiler does a better job with floating-point code in this instance. Note, however, that for other floating-point code, such as `dotf`, the Cyclone compiler is competitive.

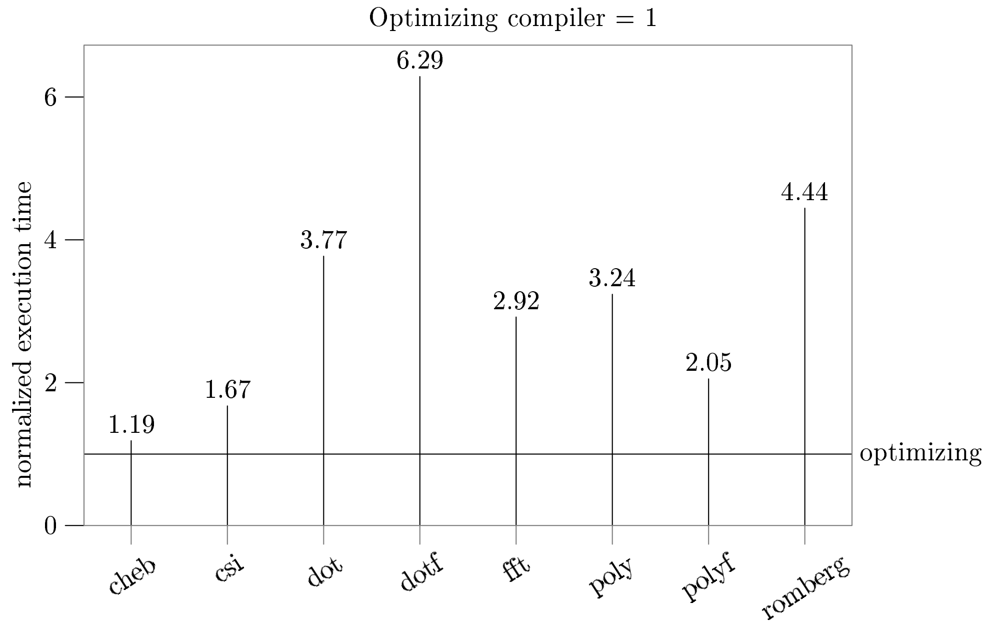


Figure 4.3: Comparing the stack-based and optimizing Cyclone compilers.

4.2 The Effect of Optimization

Figure 4.3 compares the performance of the optimizing compiler to the non-optimizing stack-based compiler for Cyclone used in previous work [HJ99]. As we expect, register allocation and the other optimizations the compiler performs are essential for achieving good performance. In every case the optimizing compiler is substantially faster than the stack-based compiler.

4.3 The Effect of RTCG

The effect of RTCG is shown in Figure 4.4 for the same input values used in the previous graphs. The execution times do not include the cost of generating the specialized code (we will come to that later). Because the specialized code takes advantage of properties of the input these results (unlike the earlier ones) are likely to depend on the input value. Figure 4.5 presents individual plots for each benchmark for a sampling of input values.

For most of these benchmarks the dependence on the input does not appear high for the inputs shown; the two exceptions are `fft` and `romberg`. In both cases the advantage of RTCG decreases as the input value (an array length) increases. This effect is due to the increased size of the generated code. Once it exceeds the size of the instruction cache, the generated code is penalized because subsequent runs must reload the cache. Under this model, we expect performance to begin deteriorating once the code size exceeds the size of the instruction cache ($16KB$ in

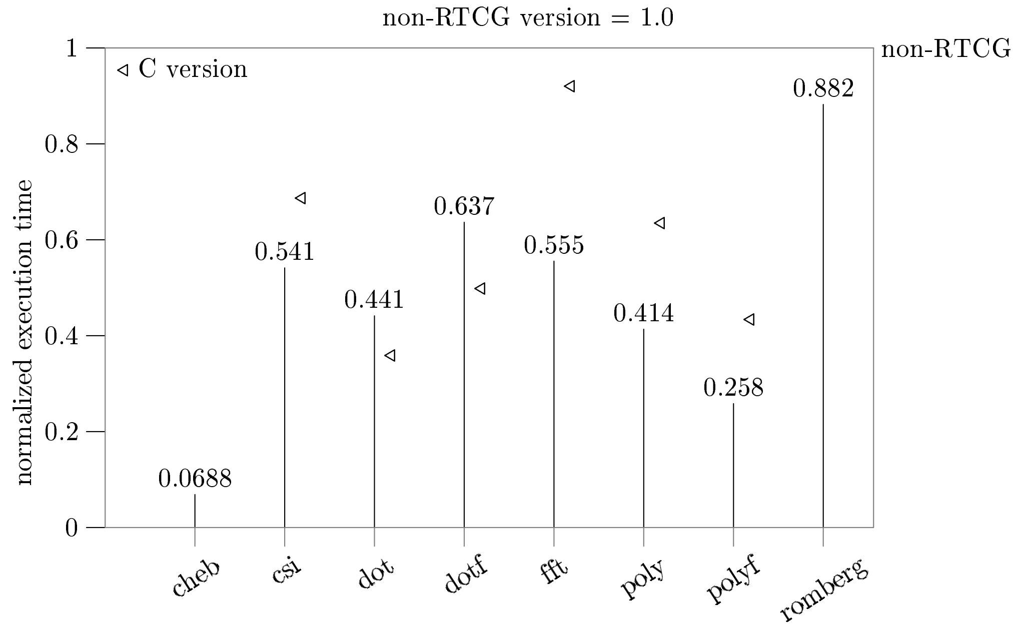


Figure 4.4: Comparison of the non-RTCG versions to the RTCG versions. (C version (◁) is included for reference when < 1.0.)

our case) and level off at twice that size. (At twice the size each instruction read misses the cache; it cannot get worse than that until the L2 cache is exhausted.) Figure 4.6 shows the performance of the specialized version of `dotf` as the length of the vector is increased. In this case, the vector contains no zeroes and cannot be optimized so the length of the generated code is linearly proportional to the length of the input vector. In the generated code, the prologue and epilogue templates have a combined size of 12 bytes. The body template consumes 38 bytes. The number of bytes for the generated code for an input vector of length L is therefore $38 \times L + 12$. As predicted, the performance starts to deteriorate at exactly $16KB$ and levels off at $32KB$.

Figure 4.7 and Figure 4.8 show the cost of code generation and the break-even points for each of the benchmarks and inputs shown in Figure 4.5. Recall that the break-even point is the minimum number of times that the specialized function must be called to beat the performance of the non-specialized version. The figure shows break-even points relative to the C version as well as the Cyclone version. The break-even points we observe, in the range 10 to 200, are small relative to the number of uses small kernels like these would see in real applications.

The break-even points for Cyclone are in many cases similar to results obtained by Tempo and DyC for similar benchmarks. DyC achieved break-even points of 2 for `cheb`, 100 for `dot`, and 31 for `romberg` [GMP+00]. Tempo achieved break-even points of 38 for `romberg` (input = 6), and 3 for `cheb` (input = 10) [NHCL96]. For `fft` Cyclone does worse. Tempo reports a break-even point of 7, and Cyclone has

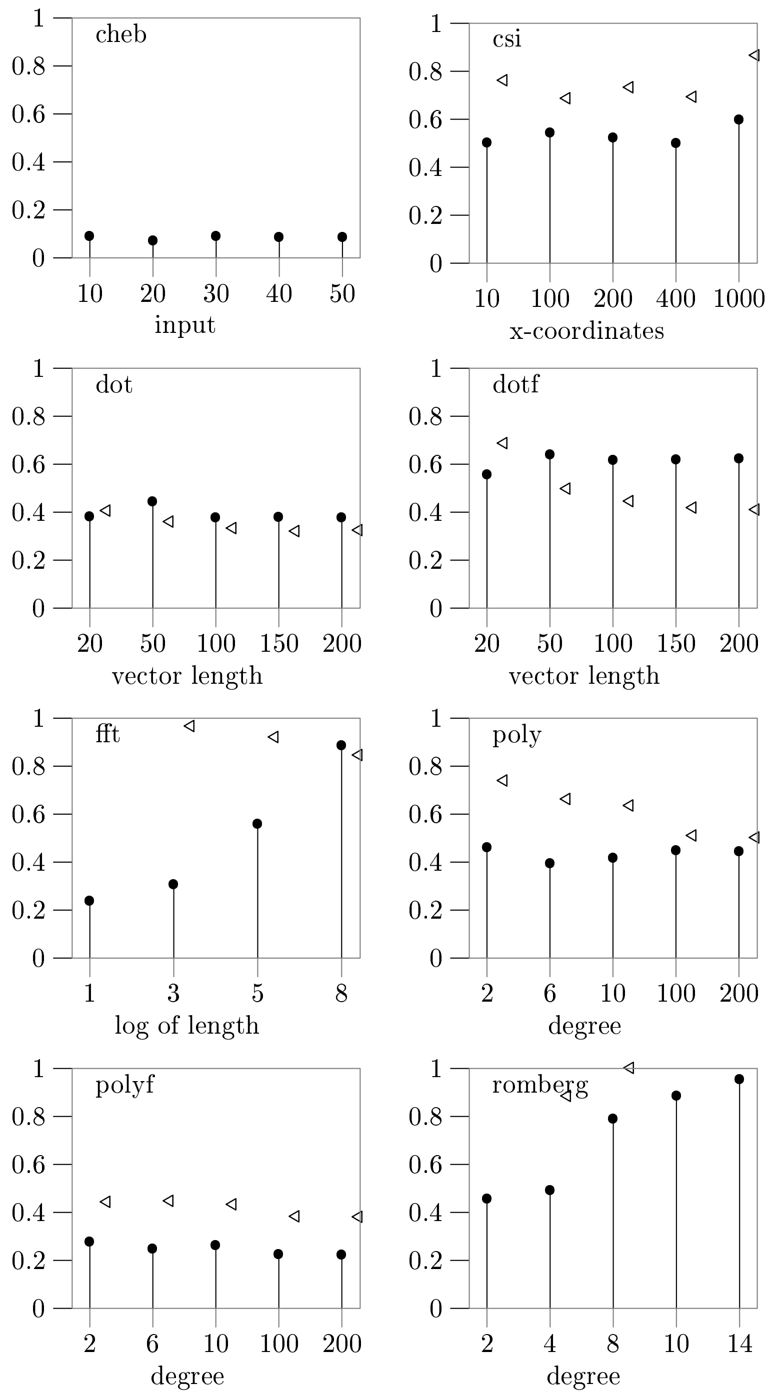


Figure 4.5: For each benchmark and various inputs, comparison of the RTCG version to the non-RTCG version (non-RTCG version = 1, unsafe C version shown with ◁ only when less than 1).

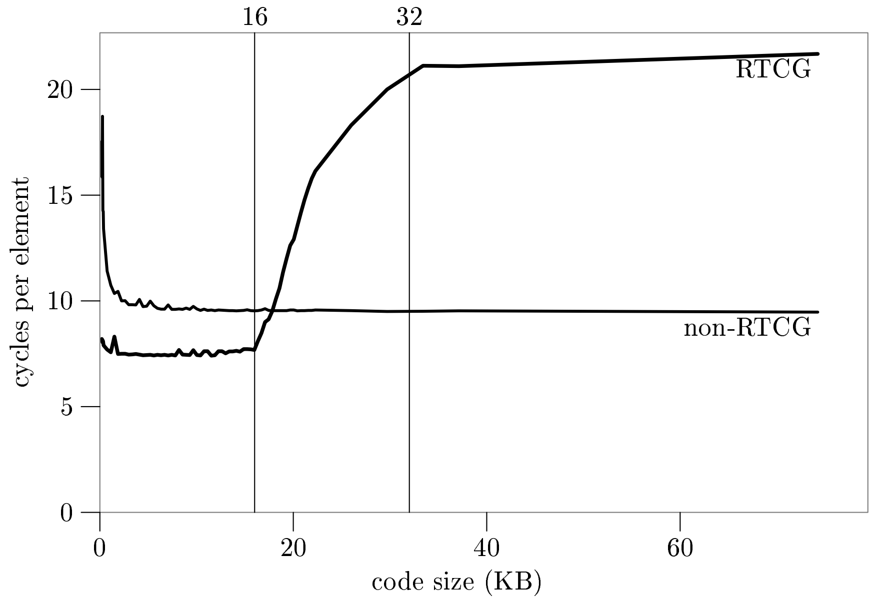


Figure 4.6: Performance of RTCG `dotf` as a function of generated code size.

a break-even point of 67. We speculate this differences may be due to the way the code was specialized and possibly array-bounds checking.

4.4 Alternatives to RTCG

In this section we consider alternatives to RTCG that exploit the same properties and see how they compare to RTCG.

For example, the performance improvement obtained by RTCG for `dot` is in part due to eliminating multiplications by zero. If the input vector is known to contain many zeroes we might choose to represent it as a linked list instead of an array. Such a sparse representation would not represent the zeroes at all. Figure 4.9 shows the performance of `dot` for a vector of length `100` as a function of the percentage of zeroes in the vector. As one would expect, the array implementation stays constant but both the specialized version and the sparse implementation drop as the number of zeroes increases. The RTCG version is slightly faster (perhaps because of eliminating loop overhead) that the sparse version but not significantly so.

Surprisingly the sparse implementation is faster in all cases than the array version. Examining the assembly files output by the compiler shows the reason. The array version's inner loop contains two arrays; each array requires two variables (one for the length, the other for the elements). In conjunction with the loop counter, all local variables can no longer be allocated to registers. Therefore, the array version performs 7 memory accesses per iteration whereas the sparse

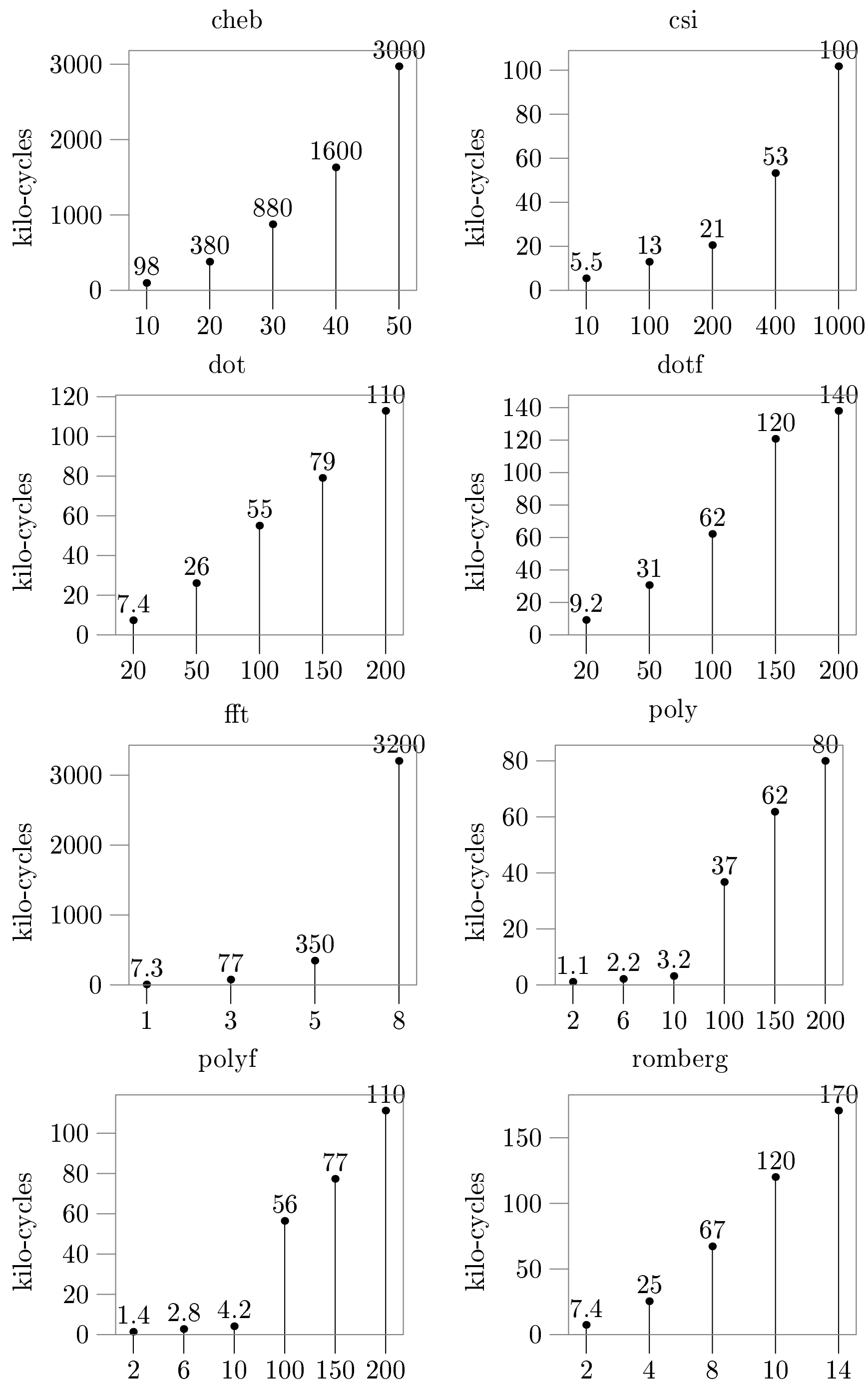


Figure 4.7: The cost of code generation in kilo-cycles for each benchmark and the same inputs used in Figure 4.5.

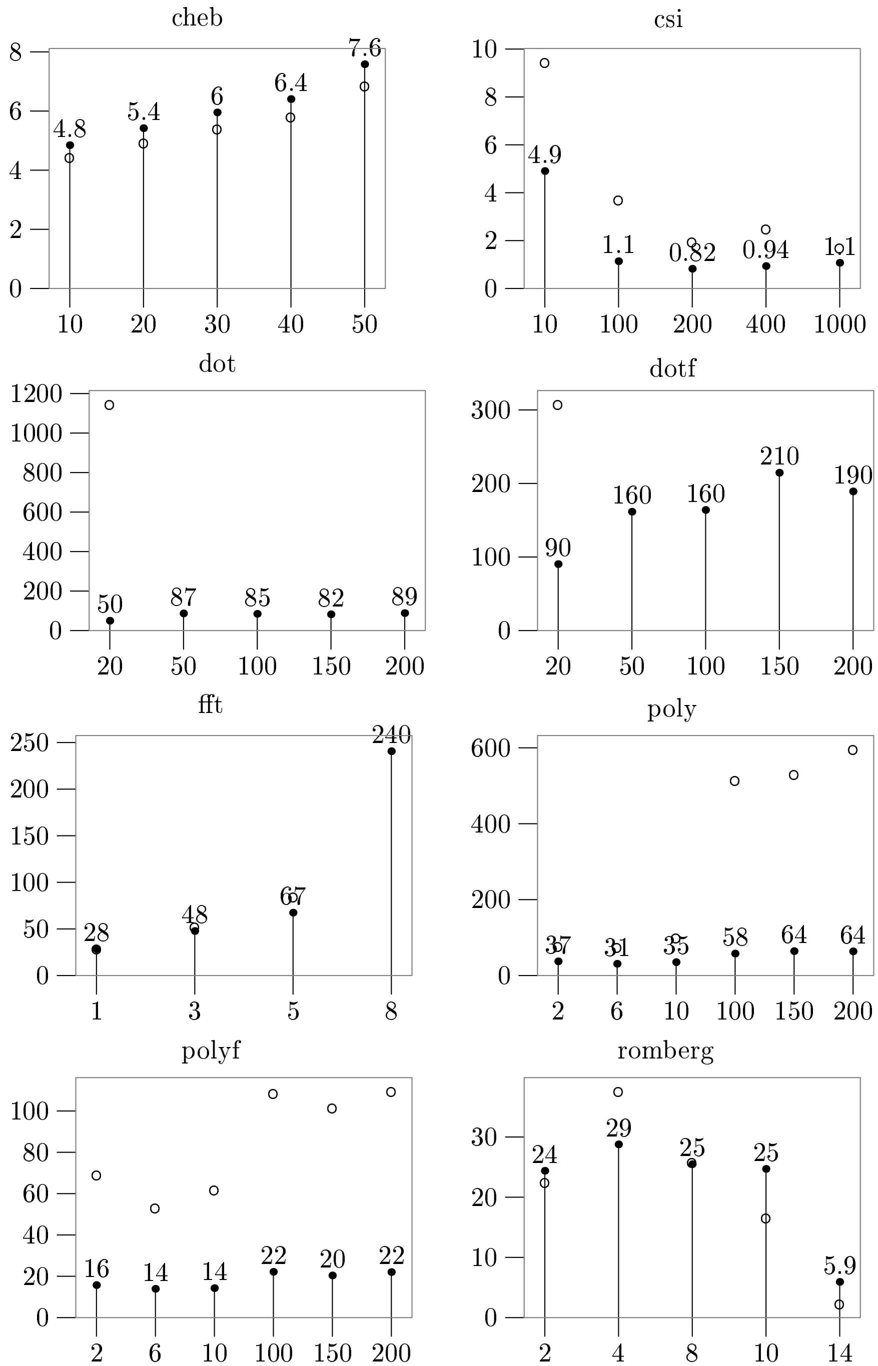


Figure 4.8: Breakeven points relative to Cyclone (labeled) and unsafe C (○) for the inputs used in Figure 4.5.

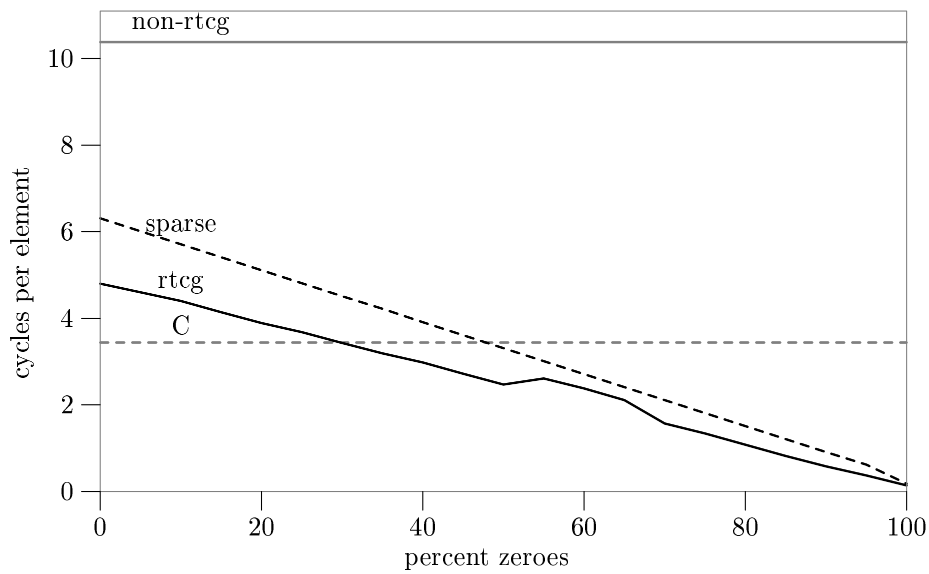


Figure 4.9: Comparison of different implementations of `dot` as a function of the number of zeroes in a 100 element input vector.

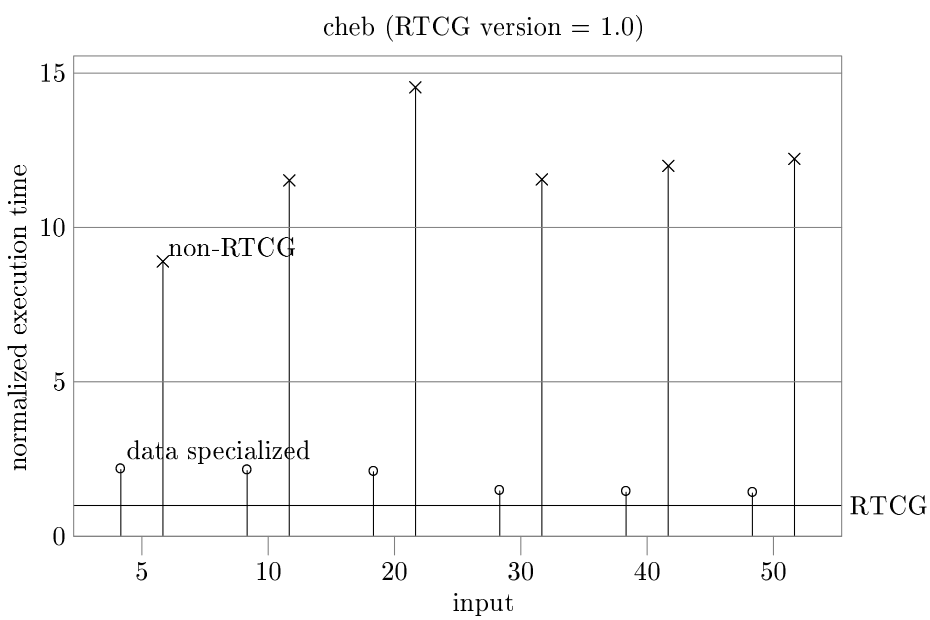


Figure 4.10: Comparing data specialization to RTCG in `cheb`.

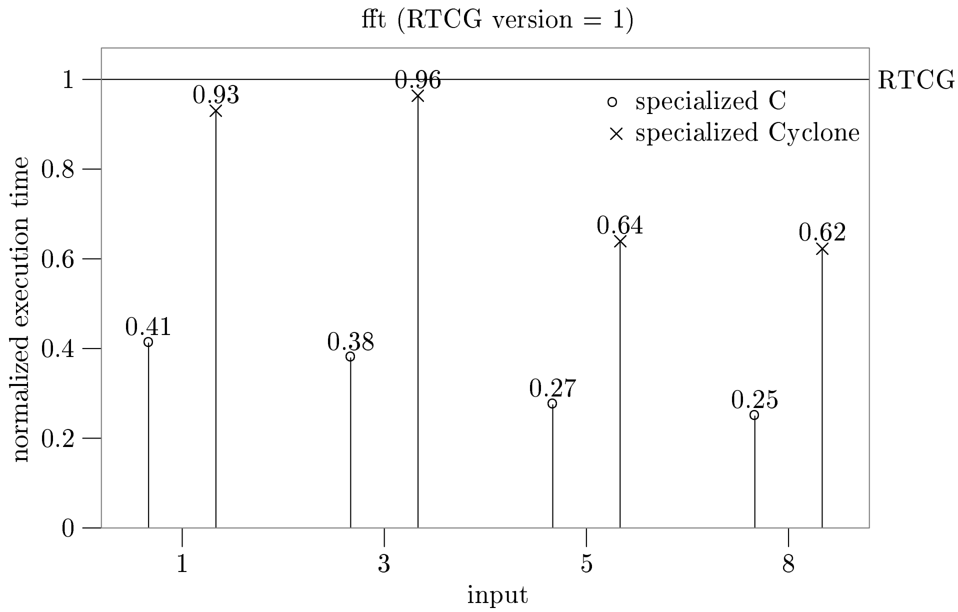


Figure 4.11: Relative performance of compile-time specialized versions of `fft` to RTCG version (two bars per input).

implementation only performs 4. This example illustrates the dangers of placing too much trust in results from micro-benchmarks.

Of the remaining benchmarks only `cheb` seemed amenable to a similar technique. In that case, pre-computing cosines and embedding the results in the generated code accounted for most of the savings. Rather than using RTCG, we could pre-compute an array with the needed cosine values. This technique of pre-computing and caching values is known as data specialization [KR96]. Figure 4.10, shows the affect of applying data specialization to `cheb`. As demonstrated by this example, when data specialization is applicable, it can duplicate many of the benefit of RTCG and is much easier to implement.

4.5 Different Kinds of RTCG

How much better could we do if we compiled the code at compile time rather than using templates? To estimate the quality of the generated code we took `fft` and wrote a C program to produce the Cyclone code (and C code) equivalent to the specialized function. Figure 4.11 compares the performance of the compile-time specialized function to the performance of the run-time specialized function. In this case, template-based RTCG is producing a function within about 40% of the performance that the Cyclone compiler is able to achieve; but more than twice as slow as the best C version.

Another important characteristic of template-based RTCG is that it is fast.

Table 4.1: Code generation time (in seconds) and break-even points for compile-time specialized fft.

	Code generation time			Break-even relative to Cyclone		
	Specialized		RTCG	Specialized		RTCG
Input	C	Cyclone		C	Cyclone	
1	0.19	0.28	0.00000974	46,542	79,091	2.8
3	0.25	0.37	0.000100	9,267	16,838	4.7
5	0.57	0.80	0.000462	4,371	8,069	6.8
8	7.8	14.55	0.00435	6,639	21,435	24.6

To compare the cost of generating code, we timed the code generation for the RTCG version and the cost of compiling the two specialized versions. For both the C and Cyclone versions we pre-processed the file before timing the compilation. The C version was compiled with `gcc -O2`. The results are shown in Table 4.1. Template-based RTCG is significantly faster and consequently has a much lower break-even point relative to the original Cyclone version of fft than either of the compile-time specialized functions.

Chapter 5

An RTCG Raytracer

The C-Mix partial evaluator has been applied to a raytracer written in C with success [And95, Mog86]. In that work, they showed an improvement in running time between two and three times. However even for small scenes containing five objects, compilation of the specialized code by the C compiler took several minutes whereas the rendering took tens of seconds [And95]. Therefore it is unlikely that anyone would choose to use offline partial evaluation for raytracing. RTCG has the potential to generate code much faster while achieving similar improvements. This chapter covers our attempt to apply RTCG to an award winning raytracer.

We used the winning entry of the ICFP 2000 programming contest as our baseline implementation. After porting the raytracer into Cyclone, we applied RTCG to the result. Using a contest-produced raytracer had several advantages over a more developed system: 1) the code base was of manageable size since it was produced by a team in only 72 hours, and 2) the code was well-implemented (it was correct and beat all other entries in terms of speed).

5.1 Overview of Raytracers and GML

This section provides a brief overview of raytracing and the GML modelling language. This exposition is intended to provide enough detail to understand the performance tradeoffs involved in the design, not enough detail to implement a raytracer. For complete details please refer to the contest documentation [RM].

The original raytracing algorithm was introduced by Whitted [Whi80] with the goal of visualizing a three-dimensional scene on a two-dimensional surface. To do so, an artificial viewpoint and a plane grid are placed in the scene. The plane represents the desired view plane (screen) and the grid cells represent the points to be sampled (pixels). To mimic the light impinging on the human retina and create the illusion of three-dimensional space, a ray-tracer tries to predict the color and intensity of the light rays passing through each pixel and hitting the viewer.

A naive solution to this simulation problem might start from each light source in the scene and follow photons from the light source through the scene, simulat-

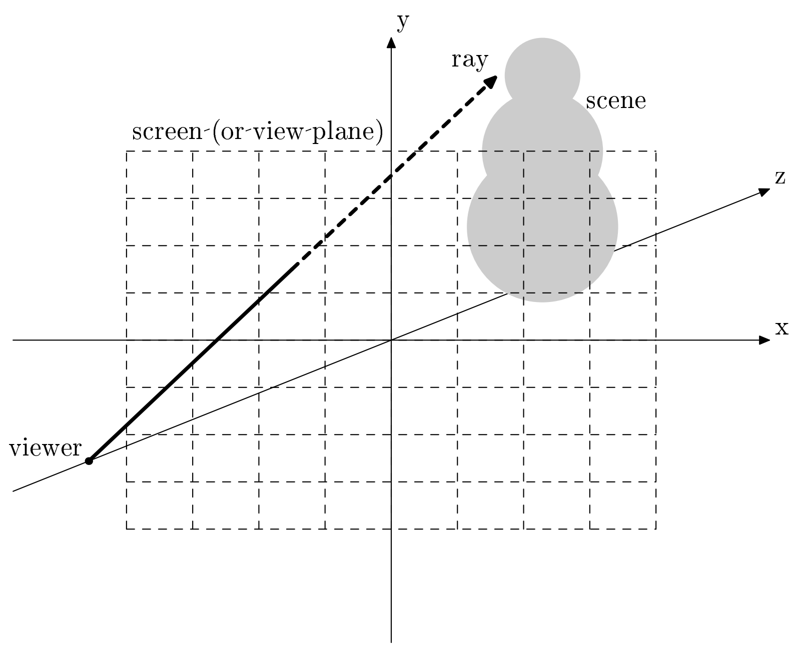


Figure 5.1: Depiction of Raytracing.

ing each physical interaction. The photons that reached the viewer and passed through the plane would be used to illuminate the screen. The difficulty with this approach is that only a miniscule fraction of the photons in the scene actually hit the viewer. A more tractable approach, and the one introduced by Whitted, runs the simulation backward. Instead of tracing photons from sources to the viewer, a ray is cast backward from the viewer through each pixel. When a ray hits an object, a shader is called to compute the surface color based on the light from sources in the scene, and any reflected component.

To compute the light incident on the surface, the raytracer casts a ray from the surface towards each light in the scene. If that ray is not obstructed then the light contributes to the intensity and color of the point. Because these rays are responsible for creating shadows in the scene, they are called shadow rays. Our raytracer supports only lights originating at a single point (pointlights). Sophisticated raytracers use area lights at greater cost and for greater realism. This feature was not required of the contest entries. For even greater realism, other raytracers also attempt to compute the effect of indirect lighting. We do not.

Reflection is incorporated into the surface color by casting a ray back along the angle of reflection according to the laws of physics: the angle of reflection equals the angle of incidence. Reflections are recursively cast until the effect is negligible or a depth cutoff is reached.

To render a scene, therefore, we require a model permitting intersection tests (to see whether we hit an object), computation of normals (to determine the angle of reflection), a specification of how a surface interacts with the incident light. We

Figure 5.2: Rendering of snowgoon.gml

use the modelling language GML to describe the scene.

In GML, scenes are represented as a single object constructed from primitive objects: spheres, cubes, cylinders, cones, and half-planes. Each primitive object has a canonical size and location. Objects are positioned and resized using affine transformations: translation, scaling (along any axis), and rotation. Interesting scenes are created by combining objects with the set operations union, difference, and intersection. (These operations are known as the CSG operations.) Figure 5.2 shows an example scene making use of many of these features.

Internally, a scene is represented as a tree whose internal nodes represent set operations, and whose leaves denote transformed primitive objects. Affine transformations are represented as four dimensional matrices. The fourth dimension allows the matrix to represent translation. To render a scene, a ray (represented as a position vector and a direction vector) is projected through this data structure, and a point of intersection is computed. The cost of computing ray-object intersections therefore usually dominates rendering time.

Ray-object intersections are computed using *interval lists*. An interval list is a list of real line segments. Each segment represents the intersection of a ray passing through an object. Abstractly, ray-scene intersection works by computing the

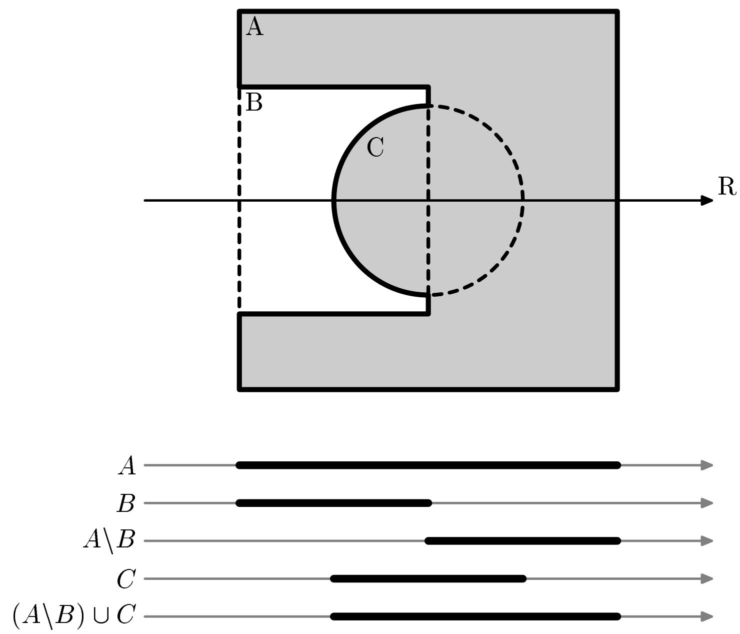


Figure 5.3: Depiction of interval list computation in 2D.

interval list for the intersection of the ray and each transformed primitive object in the scene. The resulting set of interval lists are combined via the same set operations used to construct the scene. Set operations are easily implemented for interval lists. The interval list for a transformed primitive object is conceptually computed by applying the inverse affine transformation to the ray, computing the intersection using elementary trigonometry, and applying the affine transformation to the resulting interval list.

To make this process concrete, consider the two dimensional case shown in Figure 5.3. In this figure, an incoming ray R intersects the scene $(A \backslash B) \cup C$ where A and B are rectangles and C is a circle. The visible object is shaded and has a dark outline. Edges that are not visible are shown dashed. Under the figure, we show each step in the interval list computation (the interval lists are the darkened portions.)

In practice, the above implementation can be dramatically improved. Intersections with certain primitive objects, such as planes and spheres, can be done more efficiently than described. But the biggest gains can be obtained by eliminating intersection tests with objects that will not be hit and can therefore not change the result.

Bounding volumes are used to reduce the number of ray-object intersection tests. A heuristic is used to divide the space into collections of nearby objects. Each collection is enclosed with a bounding volume as tightly as possible. Rays that do not enter a bounding volume will not intersect any object inside the volume. Therefore the raytracer checks intersection with the bounding volume before

```
{/u /v /face white 1.0 0.2 1.0}  sphere 1.5  uscale /ball1

{/u /v /face white 1.0 0.2 1.0}  sphere 2.0  uscale /ball2

{/u /v /face white 1.0 0.2 1.0}  sphere 1.1 1.3 1.0  scale /head

{/u /v /face orange 1.0 0.5 1.0}  cone
0.2 1.0 0.2  scale
90.0  rotatex
0.0 0.0 -1.0  translate
/carrot_nose

{/u /v /face black 1.0 0.0 1.0}  sphere 0.2  uscale /eye

{/u /v /face white 1.0 0.0 1.0}  sphere 0.4 0.2 0.5  scale  /scream

head
carrot_nose  0.0 -0.2 -1.0  translate  union
eye         -0.3  0.1 -0.8  translate  union
eye          0.3  0.1 -0.8  translate  union
scream       0.0 -0.5 -0.9  translate  difference
-30.0  rotatex %% Tilt the head down.
ball1        0.0 -2.5  0.0  translate  union
ball2        0.0 -5.0  0.0  translate  union
/frosty
```

Figure 5.4: Excerpt from snowgoon.gml. (Snowgoon.gml is rendered in Figure 5.2.)

examining any objects inside it. The shape of the bounding volume is chosen to make intersection tests easy while surrounding the enclosed objects tightly. Our ray-tracer uses bounding spheres. Other ray-tracers sometimes also use bounding cubes. (Octrees are bounding cubes obeying additional properties.)

Rendering can be further accelerated by specializing the ray-object intersection function. For the initial ray we only need the first point of incidence in an interval list. When computing shadow rays the exact point of incidence does not matter only whether there is one or not. In both cases optimizations are possible in portions of the scene that only use set union operations. Therefore, our ray-tracer contains a special-purpose ray-object intersection function for each of these cases.

The final component our raytracer needs is a way to specify surface properties for primitive objects. For this purpose it uses functions written in the modelling language GML. The functions take as input arguments a position on the surface, and return the color, and reflectivity at that point.

The GML modelling language is a combination of Scheme [RC91] and Postscript [Ado85]: In other words, GML is a dynamically-typed functional language with an implicit stack. Objects are specified in GML by language primitives (sphere, cone, cube, cylinder, and plane) taking a surface function (described above) as argument. Objects may then be combined with the *CSG* operations union, intersection, and difference, and transformed by translation, scaling, and rotation. In this way programmers can describe complex scenes in GML. Figure 5.4 shows an excerpt of the source code for the scene in Figure 5.2 with GML operators in italics. Curly braces $\{\ldots\}$ define a new function, and $/x$ pops the top value off the stack and binds it to the variable x.

Because each pixel requires interpreting a GML function to determine the color and reflectivity, the cost of interpretation may be high when rendering GML scenes. The winning raytracer uses de-Bruijn indices [Bru72] to accelerate variable lookup operations, and detects constant surface functions.

5.2 Opportunities for RTCG

Raytracing presents many opportunities for RTCG. For each pixel in the image a ray is cast. If that ray hits an object, yet more rays are cast to each light source. Finally, if there is a reflected component this process is iterated. Each time a ray is cast, the scene is traversed (each ray is independent of the others). For a small 400×400 pixel scene, there are 160,000 pixels and therefore at least 160,000 rays suggesting that it should be possible to recoup the cost of code generation without much difficulty. In the previous chapter we saw that break-even points of around 100 are common. Our goal is therefore to examine the ray-tracing code to discover which parts of the computation depend only on the scene.

Hierarchy. Figure 5.5 shows the source code for the function `find_all`. This function is responsible for traversing the scene and computing the resulting interval list. Clearly, if the scene is known, there is no need to test which object comes next so both switch statements can be compiled away.

Figure 5.6 shows the result of specializing the switches in `find_all`. In our specialization we have chosen to make recursive calls for bounding spheres but not for unions. Our experiments indicate that this choice is advantageous for RTCG but not otherwise. The code fragment shown in the figure is enclosed in the function in Figure 5.7. The function `find_all_gen` uses a hash table mapping objects to functions to avoid generating code for the same object more than once. Also, we special case for scenes consisting of exactly one object to break the recursion present for primitive objects (see Figure 5.6).

Ideally the specialized `find_all` need not make any recursive calls. However, we were unable to encode such a fully optimized function in Cyclone. To inline a recursive function call requires some means for extending the scope of conditionals to include the code generated by it. A simple extension to Cyclone would be to

```
interval find_all(scene sc,vector orig, vector dir) {
  // orig and dir make up the incoming ray.
  while(true) {
    switch sc {
    case SObj *(k,_,m): // m is the matrix.
      switch k {
      case SSphere *(x,r2): return sphere (orig,dir,sc,x,r2);
      case SEllipsoid: return ellipsoid(orig,dir,sc,m);
      case SCube(_): return cube (orig,dir,sc,m);
      case SCylinder(_): return cylinder (orig,dir,sc,m);
      case SCone(_): return cone (orig,dir,sc,m);
      case SPlane *(eq,_): return plane (orig,dir,sc,eq);
      }
    case SBound *(sc1,x,r2): // bounding spheres.
      if(!intersect(orig,dir,x,r2)) return null;
      sc = sc1;
    case SUnion *(sc1,sc2):
      return union_i(find_all(sc1,orig,dir),find_all(sc2,orig,dir));
    case SInter *(sc1,sc2):
      _ l1 = find_all(sc1,orig,dir);
      if(l1==null) return null;
      return inter_i(l1,find_all(sc2,orig,dir));
    case SDiff *(sc1,sc2):
      _ l1 = find_all(sc1,orig,dir);
      if(l1==null) return null;
      return diff_i(l1,find_all(sc2,orig,dir));
    }
  }
  return null; // Should never get here!!
}
```

Figure 5.5: Source for the scene traversal function `find_all`.

```
_ code = codegen(interval find_all_rtcg(vector orig, vector dir) {
  _ l = null;
  cut {
    _ scs = ^list(sc,null);
    for(;scs != null;) {
      _ sc = scs.hd;
      scs = scs.tl;
      switch sc {
        case SObj(_):
        splice { l = union_i(l,fill(find_all_gen(sc))(orig,dir)); }
        case SBound *(sc1,x,r2):
        splice {
          if(intersect(orig,dir,fill(x), fill(^(r2)).1)) {
            l = union_i(l,fill(find_all_gen(sc1))(orig,dir));
          }
        }
        case SUnion *(sc1,sc2): scs = ^list(sc1, ^list(sc2, scs));
        case SInter *(sc1,sc2):
        splice {
          _ l1 = fill(find_all_gen(sc1))(orig,dir);

          if(l1 != null) {
            l1 = inter_i(l1,fill(find_all_gen(sc2))(orig,dir));
            if(l1 != null) l = union_i(l,l1);
          }
        }
        case SDiff *(sc1,sc2):
        splice {
          _ l1 = fill(find_all_gen(sc1))(orig,dir);
          if(l1 != null) {
            l1 = diff_i(l1, fill(find_all_gen(sc2))(orig,dir));
            if(l1 != null) l = union_i(l,l1);
          }
        }
      }
    }
  }
  return l;
});
```

Figure 5.6: Specialization of switches in `find_all`.

```
< <scene,interval f(vector,vector)>Hashtable::table>Opt find_all_cache;

interval find_all_gen(scene sc)(vector,vector) {
  if(find_all_cache == null)
    find_all_cache = ^Opt(Hashtable::create(101,ptr_compare,hash));

  try {
    return Hashtable::lookup(find_all_cache.v,sc);
  } handle y { ; }

  _ code;
  switch sc {
  case SObj *(k,_,m):
    switch k {
    case SSphere *(x,r2): code = sphere_gen (sc,x,r2);
    case SEllipsoid: code = ellipsoid_gen(sc,m);
    case SCube(_): code = cube_gen (sc,m);
    case SCylinder(_): code = cylinder_gen (sc,m);
    case SCone(_): code = cone_gen (sc,m);
    case SPlane *(eq,_): code = plane_gen (sc,eq);
    }
  default:
     // Code from Figure 5.6 goes here.
  }
  Hashtable::insert(find_all_cache.v,sc,code);
  return code;
}
```

Figure 5.7: Specialization of `find_all`.

```
interval ellipsoid(vector orig, vector dir, scene sc, matrix m) {
  Matrix::vmul(vec_l1,m,orig);
  Matrix::vmul(vec_l2,m,dir);
  _ x = vec_l1;
  _ v = vec_l2;
  _ x2 = Matrix::square(x);
  _ v2 = Matrix::square(v);
  _ xv = Matrix::dot(x,v);
  _ delta = xv * xv − v2 * (x2 − 2.0);

  if(delta<=0.0) return null;

  _ sq = sqrt(delta);
  _ t1 = (−xv − sq) / v2;
  _ t2 = (−xv + sq) / v2;
  if(t2<0) return null;

  _ in = new_interval(0.0,t2,sc,sc,null);
  if(t1>0.0) in.left = t1;

  return in;
}
```

Figure 5.8: Intersection test for an ellipsoid.

allow a function to inline code-generate itself. In addition to being an ad hoc solution, TAL/T is not sufficiently polymorphic to support this extension. Our analysis indicates that this limitation did not significantly limit the optimizations we were able to achieve.

Primitive objects. In the code for `find_all` the transformations are broken out and passed to the primitive intersection functions. Notice that planes and spheres are treated specially. A sphere transformed by an isometric matrix is completely described by a center and a radius. Therefore this information is all that is needed by the intersection function. Similarly, planes remain planes under all affine transformations. Therefore their geometry is completely specified by a normal and a point.

The intersection functions for the other shapes all begin with two vector-matrix multiplies (the intersection code for an ellipsoid is shown in Figure 5.8). These multiplications transform the incoming ray into the coordinate system of the primitive object. Because we know the scene ahead of time, we can generate special-purpose code for these multiplications to avoid floating-point multiplications and additions when matrix elements are zero or one. We were unable to achieve any other savings in the intersection functions.

Lights. The lights in a scene are also known in advance. Once the ray hits an object, the top-level loop iterates through the lights in the scene and casts a shadow ray to each. The results of these tests are used to compute the color of the surface. Because each computation for a particular light is small we unroll this loop completely, and specialize on the colors and positions of the lights.

Interpreter. For surfaces with complex functions, interpreting the surface function can dominate the cost of rendering. We can compile these interpreted functions to save time.

The GML interpreter is structured as a loop taking a list of tokens as an argument. Each token is either a value in which case it is pushed on the stack, or an n-ary operator. In that case, the arguments are popped off the stack, the operator is applied, and the result, if any, pushed. In addition programs may bind variables. Evaluating a variable causes the value it represents to be pushed on the stack.

There are two control-transfer instructions in GML: `apply` and `if`. The `apply` operation reads a closure off the stack and executes it in the current context. The `if` statement pops two closures and a boolean off the stack, and executes one of the closures depending on the value of the boolean. A closure consists of a pair of a list of tokens, and an environment mapping the free-variables in the code to values.

To specialize the interpreter, we unroll its main loop. For example, Figure 5.9 shows a compacted version of the code used to evaluate the CSG operations, and the specialized version.

Most of the other operations are similarly simple, with the exception of `apply` and `if`. A naive implementation of `apply` would read a closure off the top of the stack, and call the function represented by it. Although semantically correct, this implementation strategy would be a poor choice because each function call would result in an extra stack frame being pushed on the machine stack (not the simulated GML stack). Functions are used in GML to implement loops, and would therefore likely exhaust the stack space quickly. We restructure our interpreter using trampolining to solve this problem. (Section 3.4.2 describes trampolining.)

5.3 Methodology

To measure the effects of RTCG we compared the performance of the raytracer with each of the optimizations selectively enabled for a number of scenes.

All experiments were performed on a 750Mhz Penium III with a 256-KB L2-cache and 256 MB of memory running Linux Redhat 7.0. Overall running times were measured using the C `getrusage` function. Each run was repeated at least

Non-RTCG:

```
case Obj2(op):
   _ o1,o2;
   POP_OBJ(o2); // macro for stack pop.
   POP_OBJ(o1); // POP_OBJ checks that the values are objects.
   _ o;
   switch op {
   case Difference: o = ^obj.Diff(^(o1,o2));
   case Intersect: o = ^obj.Inter(^(o1,o2));
   case Union: o = ^obj.Union(^(o1,o2));
   }
   PUSH(^value.Vobj(o)); // Stack push macro.
```

RTCG:

```
case Obj2(op):
   splice {
      _ o1,o2;
      POP_OBJ(o2);
      POP_OBJ(o1);
      _ o = o1;
      cut {
         switch op {
            case Difference: splice o = ^obj.Diff(^(o1,o2));
            case Intersect: splice o = ^obj.Inter(^(o1,o2));
            case Union: splice o = ^obj.Union(^(o1,o2));
         }
      }
     PUSH(^value.Vobj(o));
   }
```

Figure 5.9: Cyclone code implementing the GML CSG operations.

Table 5.1: Scene statistics

	primitives	bounding spheres	lights	pixels	baseline(seconds)
chess	261	101	1	307200	17.68
dice	46	32	1	786432	5.55
fractal	586	435	2	307200	18.23
go	532	489	2	307200	55.34
kal	7	4	1	160000	12.22
mandel	2	0	1	64000	54.49
snowgoon	33	29	2	786432	13.11
texture	2	0	1	360000	64.00
tree	364	201	1	460800	10.84

three times and the best time taken. When great variance was observed more runs were used. To measure garbage-collection time and code-generation time we instrumented the garbage collector and run-time libraries. We relied on the processor's cycle counter to measure the time spent therein. We also instrumented these two libraries to collect information about the amount of allocation, the amount of code generated, and the number of functions generated. Separate runs were used to collect this auxiliary information. The instrumented libraries were not used when measuring end-to-end performance.

We chose nine scenes designed to exercise the raytracer, and of sufficient complexity to be interesting. Table 5.3 describes the key characteristics of each scene. The scene sizes were chosen so that the scenes could be rendered by the baseline Cyclone implementation in more than 5 seconds and less than 1 minute on our test machine. Many of these scenes were submitted to the contest officials as part of the competition:

1. chess [KR] won the judges prize for Leif Kornstaedt and Andreas Rossberg in the competition. The scene depicts a chess board at the beginning of a game. Each piece is composed of multiple objects.

2. dice was designed by John Reppy. It was one of the sample scenes given to contestants. The scene shows two dice suspended in mid-air over a flat grey plane, before a mirror. The spots on the dice are cut-out using set difference with spheres.

3. fractal [DHa] is a scene by Damien Doligez and Robert Harley. This scenes shows a collection of colored reflecting spheres arranged in a fractal pattern over a distant blue plane. I changed the viewpoint to bring the collection closer and increase the number of visible reflections.

4. go [DHb] was designed by Damien Doligez and Pascal Cuoq. This scene shows a go game in progress. Each go piece is either a white or black ellipsoid. The board has a realistic wood texture that is computed by programmer supplied surface functions.

5. kal [DHc], which stands for kaleidescope, contains only a small number of objects but a high number of reflections. The scene consists of three reflective planes arranged to form a three-dimensional triangle with one open side facing the viewer. A sphere and a cylinder placed between the planes are reflected like beads in a kaleidoscope. It was designed by Alan Schmitt.

6. mandel is a scene consisting of a sphere with a surface composed of the Mandelbrot set sitting on a plane.

7. snowgoon is a scene I adapted from a Bill Waterson cartoon [Wat92]. It is shown in Figure 5.2.

8. texture uses a surface combinator library to place an interesting texture on a sphere. The scene includes only the sphere with a swirling plaid texture on it.

9. tree uses a three-dimensional fractal algorithm to generate a tree. The tree body and limbs consist of cylinders. The ends of the trees are made up of balls. The scene contains no reflective elements but many shadows.

Of these scenes only go, mandel, and texture spend a significant fraction of their time shading surfaces as opposed to tracing rays. I created mandel and texture especifically to test the effectiveness of generating code for the interpreter.

Our baseline implementation of the raytracer in Cyclone is almost a direct port of the winning raytracer. We implemented one algorithmic optimization in the OCaml raytracer before porting it: automatic depth cutoff. This optimization stops iterating on reflection when the effect of reflection cannot effect the outcome. This proved significant in speeding up some of the highly reflective scenes.

Our initial implementation was substantially slower than the OCaml version. By profiling the code we discovered that the problem was excessive allocation. The initial implementation allocated new vectors for every vector matrix operation, and allocated new intervals for each interval list computation. We modified the code to avoid allocation when possible, and pre-allocated a large number of intervals – twice the number of primitive objects in the scenes if less than a preset upper limit. The only place we were not able to cut allocation was in the evaluation loop of the interpreter. Figure 5.10 compares the performance of the Cyclone implementation to the OCaml entry.

Profiling data shows that the Cyclone version is slower for go, mandel, and texture because of allocation that we cannot eliminate. The Cyclone garbage collector is conservative, and relies on a function call for each allocation. In contrast the OCaml collector is accurate, optimized for short-lived data, and inlines the allocation sequence. We are not sure why the Cyclone implementation is slower for kal. The other scenes are faster due to our allocation optimization, and because we inlined the vector subtract in the intersection test for bounding spheres.

Profiling data was collected for the OCaml version using `gprof` [GKM82]. We wrote our own profiler for Cyclone that relies on the processor's time-stamp

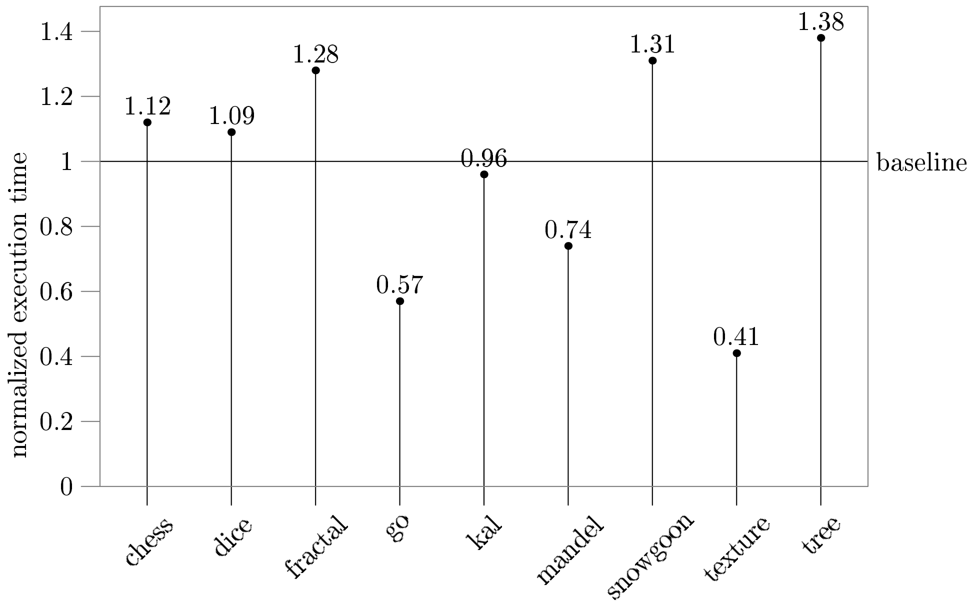


Figure 5.10: Comparison of OCaml version to baseline Cyclone implementation.

	chess	dice	fractal	go	kal	mandel	goon	texture	tree
evaluation	2.6	0.0	0.0	27.9	0.0	56.4	0.0	90.0	0.0
find_all	11.7	5.8	6.4	10.3	8.7	0.0	8.3	0.3	8.4
gc	0.5	0.0	0.4	2.3	0.7	42.7	0.0	2.9	0.6
hit	2.0	1.7	4.2	0.0	0.7	0.0	2.1	0.0	1.6
intersect	30.3	23.0	44.1	45.2	19.6	0.0	32.8	0.0	34.3
intervals	3.3	3.1	1.0	1.5	10.9	0.1	1.6	0.0	2.0
main	1.1	4.7	0.7	0.9	0.0	0.0	2.7	0.4	1.3
math	2.8	9.7	4.5	1.7	8.3	0.1	4.5	1.7	2.4
primitives	14.1	19.7	8.8	3.5	24.5	0.1	17.3	1.3	10.8
scene_opt	0.0	0.0	24.5	0.0	0.0	0.0	0.0	0.0	8.0
trace	2.7	27.7	4.0	1.9	16.7	0.1	7.8	1.5	2.1
vmul	25.1	2.3	0.2	1.7	5.7	0.0	19.7	0.0	27.1

Figure 5.11: Breakdown of running time of the Ocaml version according to `gprof` (percentages).

	chess	dice	fractal	go	kal	mandel	goon	texture	tree
evaluation	4.5	2.7	0.4	25.6	2.4	54.4	0.9	53.6	0.0
find_all	24.4	10.8	14.7	10.9	18.0	0.1	20.0	0.3	20.9
gc	5.2	4.5	1.0	39.0	3.5	43.3	1.9	41.3	1.1
hit	4.1	5.4	8.2	0.3	1.0	0.2	4.2	0.2	5.5
intersect	15.7	8.9	22.6	12.6	7.3	0.0	15.5	0.0	19.0
intervals	9.0	6.0	3.4	2.9	15.1	0.1	7.5	0.0	4.9
main	0.0	4.2	0.0	0.0	0.0	0.1	0.9	0.4	1.5
math	0.0	2.1	0.0	0.0	0.0	0.0	0.0	0.0	0.7
primitives	15.3	12.1	6.2	1.8	20.6	0.2	15.4	0.5	12.9
scene_opt	0.0	0.0	29.4	0.0	0.0	0.0	0.0	0.0	11.4
trace	2.2	12.5	2.5	1.2	7.5	0.4	5.9	0.9	1.3
vector	2.9	21.1	8.3	1.3	16.6	0.3	12.1	1.0	2.9
vmul	10.2	2.9	0.4	0.8	3.3	0.0	8.0	0.0	14.3

Figure 5.12: Collated profiling data for the baseline Cyclone raytracer (percentages). (Data collected with our own profiler.)

counter. Our profiling technique perturbs the test program more than a sampling-based profiler like `gprof` and therefore yields less accurate results. Because the Cyclone and OCaml implementations are so similar, we use both sets of profiling data to predict Cyclone program behavior. Using the per-function profiling data, we grouped functions into categories to produce Figure 5.11 and Figure 5.12.

5.4 Effect of RTCG

This section examines the effect of each of the optimizations on the time to render each of our nine scenes. In every case code generation time was measured to be less than two-tenths of a second, and therefore did not significantly affect any of these results. All the programs fit easily into the memory of the test machine, so paging was not a factor even for our largest scenes.

5.4.1 Interpreter

Figure 5.13 shows the effect of specializing the interpreter. The three scenes that spend the most time in the interpreter; go, mandel, and texture; all are significantly slower after optimization. Even snowgoon which did not originally spend much time in the interpreter is slower.

Two effects account for this behavior: 1) the savings we obtained were relatively minor, and 2) the instruction cache penalties were large. First, because GML is dynamically typed each value carries a tag indicating its type. To implement a simple integer addition the interpreter must first read two values off the stack, check that each is indeed an integer, perform the addition, allocate a new integer

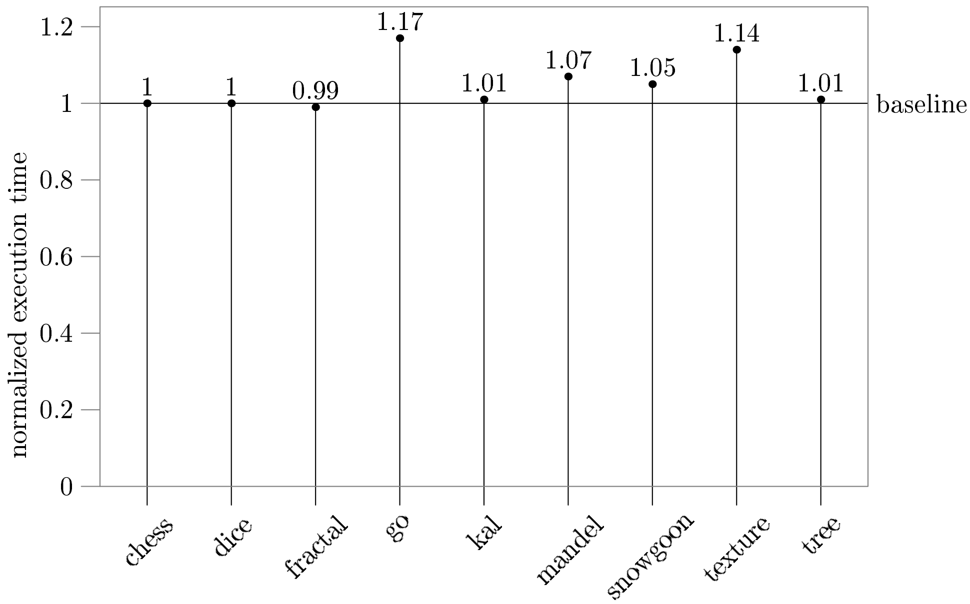


Figure 5.13: Performance of version using specialized interpreter compared to baseline.

value, and store the result on the stack. None of this cost can be easily eliminated by RTCG. The loop overhead, and the cost of fetching the next instruction have been eliminated, but these cost are small compared to the tag tests. In addition, RTCG has unrolled this loop causing the generated code to be substantially larger. For go, Cyclone generates 1.5 MB of code. Even snowgoon requires 141 KB. Since most of this generated code is identical to the original interpreted version, the only effect of all this copying is to worsen the instruction cache behavior.

One can imagine more aggressive RTCG approaches that could yield benefits. For example, we could perform an analysis to eliminate tag tests (these have been studied extensively for Scheme [WC97]), maintain separate allocation areas for each kind of value, and implement our own garbage collector. Then, RTCG might payoff. We could also compile GML into a lower level language and optimize that. In the next chapter we will see that improvements are possible for very low-level languages. In either case, the work required is essentially equivalent to implementing a GML compiler.

Other ways of implementing RTCG might yield better results in this example. A system like 'C could optimize the straight-line code once it was emitted. However, the optimizer would by necessity work in a purely local context and could not therefore propagate information across function boundaries. Without inter-procedural information, tag test elimination will probably not be effective.

Even if GML were strongly typed Cyclone's current type system could not verify that fact. Therefore, the generated code would have to perform tag tests even though they would be completely unnecessary. A type system using type

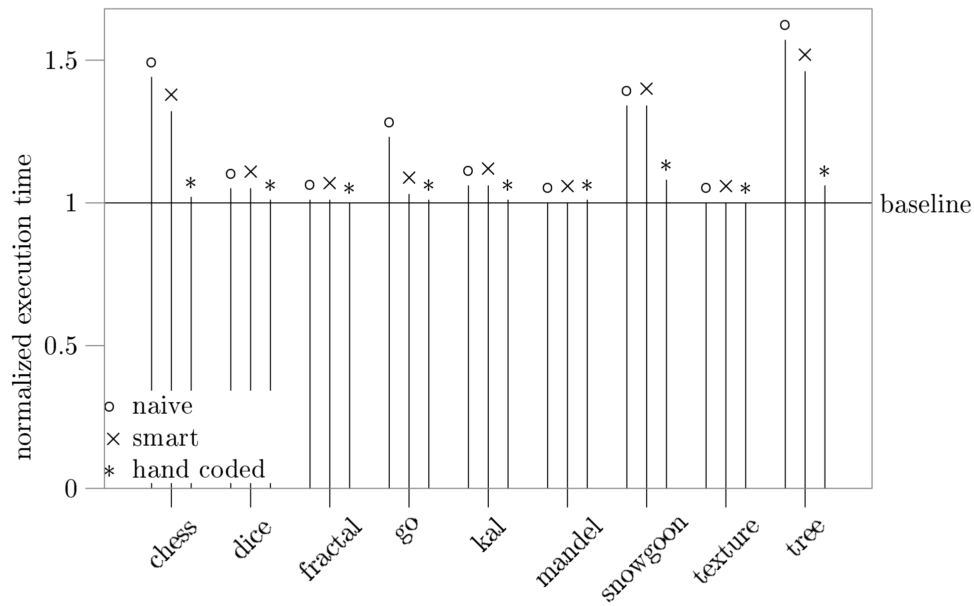


Figure 5.14: Performance of version specializing vector-matrix multiplies relative to baseline.

analysis might be able to encode these invariants and permit safe tag elimination in these cases.

These considerations lead us to believe that lightweight type-safe RTCG is not well suited for compiling high-level languages. And RTCG, without substantial dynamic optimization, is not well suited to compiling dynamically typed languages.

5.4.2 Primitive objects

This optimization places a special-purpose vector-matrix multiplication function and a specialized normal function in each primitive object (except spheres and planes). Figure 5.14 shows three versions of this optimization. The first version generates a new function for every vector-matrix multiply independently. The second version generates a new function based on the structure of the matrix, and not its values. For example if two matrices share zeroes and ones in the same locations they will use the same generated function. Our initial guess was that the poor performance was again due to the instruction cache. For the scenes with many objects, this change reduced the number of functions generated by one third to one half, and the size of the generated code by a factor of eight in the case of go. This explains why go became so much faster between the two runs.

Still, the performance is surprisingly bad. Inspecting the generated code reveals that the code generated for the RTCG case was poorly register allocated due to the large number of blocks present in the RTCG version. A better implementation of floating-point register allocation would have done better. A nice feature of TAL/T

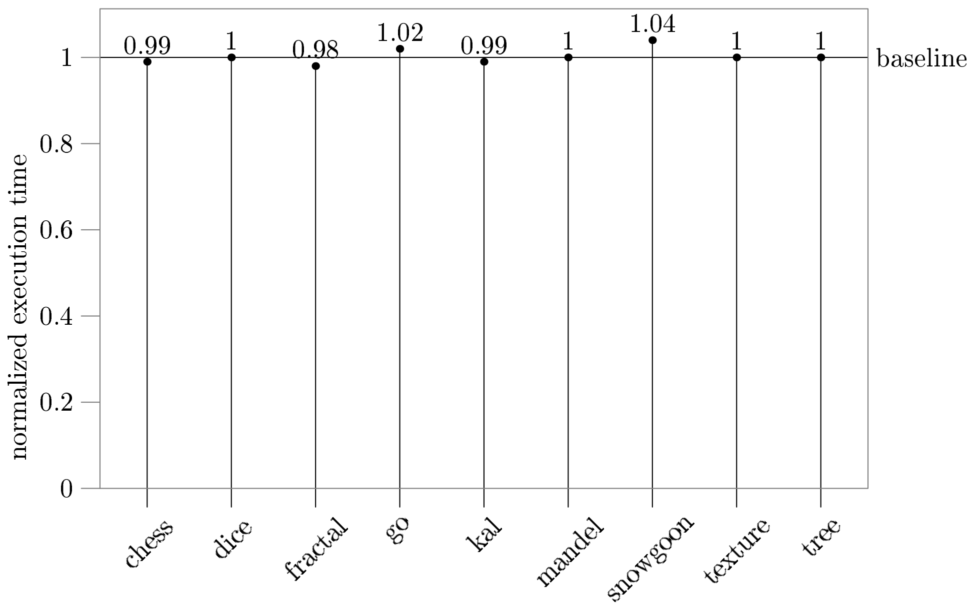


Figure 5.15: Performance of version specializing on the lights relative to baseline.

is that we can modify the assembly language by hand while retaining safety. The third bar in the graph shows the result of hand optimizing the assembly as we imagine a better compiler might.

Even with this optimization, we don't get any improvement over the baseline. The reason is that there are not very many zeroes and ones in the matrices. For the matrices permitting any optimization, we can most often eliminate less than eight multiplies. Usually, this function takes 16 additions and 16 multiplications. Therefore, the potential savings is less than half for these functions. The profile data shows that less than 10% of run time is spent in vector multiply on average. Therefore the maximum savings possible is 5%. The additional code we are generating may negatively impact the instruction cache in unpredictable ways. The observed performance is consistent with this assessment.

5.4.3 Lights

The light computations do not make up a significant fraction of run time so we do not expect this optimization to succeed. In fact as shown in Figure 5.15, the effects are inconsequential.

5.4.4 Hierarchy

This optimization specializes on the structure of the scene. Figure 5.16 shows the effect of four different versions of this optimization. The first version is the naive

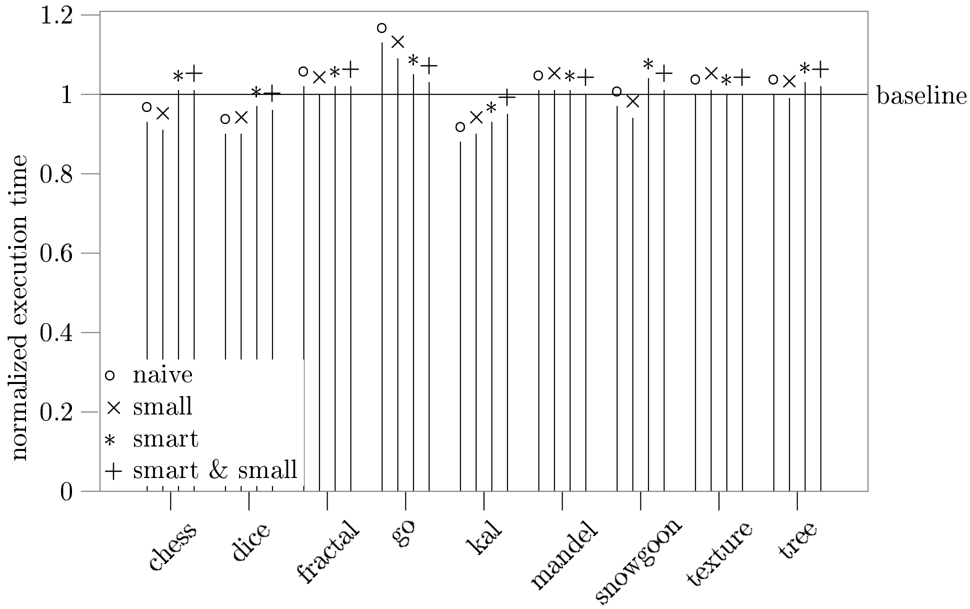


Figure 5.16: Comparison of versions specializing **find_all** to the baseline.

implementation. We see that certain scenes; chess, dice, and kal; receive a 10% performance boost. Go is slowed by almost 20%.

The function `find_all` is highly recursive. If the size of the functions generated from it exceed the instruction cache size then we expect performance to be reduced. Figure 5.17 shows the size of generated code and the size of allocated code regions. The ratio between these two values is between two and three. Since the Cyclone run-time system uses a doubling strategy for the code regions, this graph implies that the functions we are generating are on average smaller than the smallest code region. In the second column of Figure 5.16 (and Figure 5.17 we show the effect of cutting the initial code buffer size by four to 128 bytes. This change improves the performance of go but it is still worse than the baseline.

The go scene contains hundreds of objects on the playing board. Each object is an ellipsoid surrounded by a bounding sphere. Each one of these bounding spheres causes a function to be generated, easily blowing the instruction cache. Therefore, it makes sense not to generate a specialized function when very little gain is likely. The version of `find_all` shown in Column 3 of Figure 5.16 uses a heuristic to control code generation. This heuristic estimates the amount of code generated and cuts off when a preset value is exceeded. We tuned this value as best we could. Column four shows the effect of combining the two changes from columns two and three.

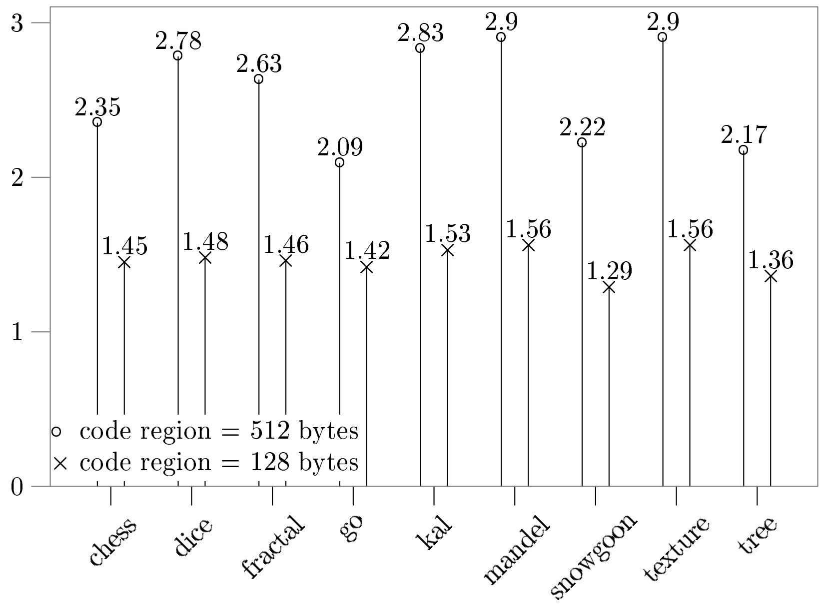


Figure 5.17: Ratio of allocated space for code region to size of generated code for different initial code region sizes. (Naive `find_all` specialization)

5.5 Conclusion

Our experience suggests that raytracing is not a good application for RTCG. The optimizations we can apply only affect a small fraction of run time, and scale badly with scene size. This result directly contradicts previous work [And95]. In that work, the authors achieved a factor of two or three improvement.

One reason for this difference is algorithmic. Their raytracer did not support union, difference, or intersection. It only allowed spheres, cubes, and discs to be placed in the scene. This change greatly simplifies the inner loop. In addition, they did not implement any bounding volumes. Bounding volumes significantly decrease the number of ray-object intersection tests. Disabling this optimization makes our raytracer run about an order of magnitude more slowly. This change probably accentuates the effect of modest improvements in intersection tests. Also, the scenes they used to evaluate their system contained between 5 and 36 objects (most of them unit spheres) scattered in the xz-plane. Objects with these characteristics have many zeroes and ones and are therefore easier to optimize.

Finally, their system passed the generated code to the C compiler. Optimizations performed offline by the C compiler may be responsible for the observed gains.

We have presented this work as an example of an application that seems to have all the prerequisites for RTCG but receives no benefit from RTCG. The literature in this area is devoid of such examples creating the false impression that RTCG always works. To avoid being overly optimistic about RTCG it is important to

1) start from the best available algorithm, 2) realistically assess the proportion of run time that can be affected by RTCG, 3) ensure that realistic data contains patterns that RTCG can exploit, and 4) check that this application of RTCG does not require generating so much code that instruction cache effects will nullify all gains.

Chapter 6

An RTCG Simulator

This chapter presents a detailed analysis and evaluation of a successful application of run-time code generation to a mid-sized program. The application we chose is the fast functional simulator from the SimpleScalar Tool Set [BAB96].

The SimpleScalar Tool Set is a family of simulators for a 64-bit architecture based on the MIPS (the SimpleScalar architecture). These simulators are widely used in the computer architecture community to study the design of memory subsystems. We cite here a small sample of relevant work [PFP99, SAMC98, LRYT99, FB99]. Each of the simulators models memory with different degrees of precision. Along with the SimpleScalar Tool Set, there are ports of a Fortran and a C compiler that can be used to generate executables for the SimpleScalar architecture.

Optimizing the simulators in the SimpleScalar Tool Set is worthwhile because simulator runs are often time-bound. On our test machine using the fastest simulator from the ToolSet, running the full SPEC CPU95 benchmark suite (a standard benchmark widely used to evaluate processor performance) would take by extrapolation about one week, although we have not tried it. More precise modelling of the memory hierarchy would require even more time.

One way to achieve increased performance would be to compile the simulated program to the native architecture. A number of instruction set simulators do take this approach [CK94, WR96]. However, by using binary translation these simulators lose two key properties: simplicity and portability. Hardware researchers regularly modify the simulators to experiment with alternative memory structures or new instructions. Binary translators are notoriously difficult to engineer, and debug; making them hard to modify as well.

RTCG provides an alternative that (in principle) combines the simplicity and architecture neutrality of an interpreter with some of the performance benefits of binary translation. Like an interpreter, it is straightforward to add instructions, instrumentation, or a memory model.

In the remainder of this chapter, we analyze our example simulator, `sim-fast` (the fastest simulator in the ToolSet), to discover what opportunities there are for optimization by RTCG. After presenting our analysis, we explain how to implement an RTCG version of `sim-fast`. Finally we evaluate the RTCG implementation using the SPEC CPU95 benchmarks.

6.1 Implementation Overview

The SimpleScalar architecture is a simplified version of the MIPS using a 64-bit instruction encoding. The architects chose an intentionally bloated instruction encoding for two reasons: to simplify decoding (the opcode is always the eight lowest-order bits), and to simplify encodings (researchers often encode additional information in unused portions of instructions). The simulator is composed of three key components: the registers, the memory, and the interpreter.

Like the MIPS, the SimpleScalar machine has a bank of 32 general-purpose registers and a bank of 32 floating-point registers, in addition to 7 special-purpose registers. The only special-purpose register pertinent to this discussion is the program counter, `pc`. The `pc` contains the address of the next instruction to execute. All registers are 32-bits wide. To support 64-bit floating point computation, the floating-point registers are reused. The double-precision instructions use the even-numbered registers to manipulate the 64-bit floating-point values obtained by concatenating the values in adjacent 32-bit registers.

The simulator uses a data structure with one entry per special-purpose register, and one array per bank of registers. In the main loop of the interpreter, the `pc` is represented by a local variable `cpc`, and the arrays of registers are placed in two local variables: `fp` for the floating-point registers, and `gp` for the general-purpose registers.

The SimpleScalar architecture supports a flat 32-bit memory model. Internally the simulator divides this space into 64KB pages. For a given memory address, the highest 14 bits are the *page address*, and the lowest 18 bits are the *page offset*. The simulator maintains an array, called the page table, with one pointer per page address (2^{14} pointers). Initially all these pointers are zero. The first time the simulated program attempts to read or write a page the entry in the page table will be zero. In this case the simulator allocates a new page and places a pointer to it in the table.[1] When the page table entry is non-zero, the sum of the entry and the page offset yields the address to be read or written.

The interpreter is invoked after the executable has been loaded into the simulated memory, and the state of the simulated machine has been initialized. Initialization includes setting the `pc` to the address of the first instruction to execute. The body of the interpreter is an infinite while-loop shown in Figure 6.1. The variable `cpc` represents the current value of the register `pc` and `npc` represents the value it will have after executing the next instruction. If the current instruction is a jump instruction, the value of `npc` will be changed in the body of the switch statement shown in the figure. The assignment to `gp[0]` is needed to ensure that this register is always zero, a requirement of the architecture.

[1] To run the CPU95 benchmarks allocate-on-read semantics are required.

```
while(true) {
  cpc = npc;
  npc = cpc + 8; // Set the next pc to point to the next instruction.
  gp[0] = 0;      // Re-establish invariant that gp[0] == 0.

  inst = READ_INST_AT(cpc); // Read the instruction to execute
  opcode = OPCODE(inst);

  switch opcode {
    // One case per instruction.
    // ...
  }
}
```

Figure 6.1: Skeleton of simulator's main loop.

RTCG can be used to optimize the interpreter by unrolling this loop and generating special purpose code for the instructions encountered.

6.2 Implementation for RTCG

To achieve the maximum benefit from RTCG we need an efficient code generation strategy. This section presents the strategy used in our implementation and explains some of the alternatives.

Figure 6.2 shows the Cyclone source for simulating the ADD instruction with and without RTCG. The two code fragments are nearly identical except for the addition of RTCG constructs. Notice that all the array accesses and instruction decoding happen at code-generation time. (The expression inside a fill is computed at code-generation time and the result is used to fill a hole in a template.) The general-purpose registers are treated differently in the two versions. In the plain version, gp refers to an array of unsigned integers. In the RTCG version, gp refers to an array of pointers to unsigned integers. By inserting an extra level of indirection, we are able to eliminate the array-bounds check in favor of a tuple access. Without this level of indirection, the array-bounds check would be necessary each time the generated code was executed. (Section 3.4.3 describes this technique.)

Figure 6.3 shows the template generated for the RTCG version of ADD. This template takes seven instructions. It might be possible to compact this template by placing the holes in the other instructions but our implementation only supports holes appearing in MOV instructions. (TAL/T uses the pseudo-instruction CGHOLE to denote a hole in a MOV.)

Most of the remaining instructions; integer instructions, memory instructions,

```
//PLAIN:
  r2 = gp[inst_b >> 24];
  r3 = gp[(inst_b >> 16) & 0xff];
  gp[(inst_b >> 16) & 0xff] = r2 + r3;

//RTCG:
splice {
  r2 = fill(gp[inst_b >> 24]).1;
  r3 = fill(gp[(inst_b >> 16) & 0xff]).1;
  fill(gp[(inst_b >> 16) & 0xff]).1 = r2 + r3;
}
```

Figure 6.2: Cyclone code for ADD with and without RTCG.

```
CGHOLE  EAX,T$7086,T$7086$558
MOV     EAX,DWORD PTR [EAX+0]  ; EAX = r2
CGHOLE  ECX,T$7086,T$7086$559
MOV     ECX,DWORD PTR [ECX+0]  ; EAX = r3
CGHOLE  EDX,T$7086,T$7086$560
ADD     ECX,EAX                ; ECX = r2 + r3
MOV     DWORD PTR [EDX+0],ECX  ; r1  = r2 + r3
```

Figure 6.3: Template for the ADD instruction.

and floating-point instructions; are similarly implemented. The control instructions require some care. Because control-flow in the simulated program is arbitrary, and control-flow in Cyclone is structured; it may seem impossible to generate code for arbitrary jumps. However, by using the arbitrary control-flow between functions, we can simulate the arbitrary control-flow between basic blocks. For each basic block in the simulated program, the RTCG version generates a function. This amounts to unrolling the main loop of the interpreter until the first control-flow instruction is encountered, at which point the interpreter is restarted.

The simulator maintains a *code table* mapping addresses to functions. To simulate a jump instruction, the code generator retrieves and calls the function associated with the target address. In the event that code has not been generated for the target address already, a new function must be generated and added to the code table.

As described, this algorithm does not correctly handle cycles between blocks. To see why, consider a block beginning at address a and ending in a jump to address a. When generating code for the final jump instruction, the simulator would lookup the code for a. Because the code has not yet been generated, the simulator would start generating the block for address a again; thus entering an infinite loop.

An extra level of indirection in the code table is the solution. Instead of mapping addresses to functions, the code table maps addresses to *block descriptors*. A block descriptor is a data structure containing a function pointer. Before generating any code, a block descriptor, with a dummy function pointer, is allocated for the new address and installed in the code table. After code generation is complete, the block descriptor's function-pointer field is updated with the newly generated function. Jumps are also modified to always indirect through the block descriptor. In the self-looping block example, retrieving the jump target will simply return a valid block descriptor instead of generating more code.

The extra-level of indirection afforded by the block descriptors allows generated code to be replaced dynamically. The implementation uses this ability to delay generating code until a block is actually executed. A translation of a jump does not cause code to be generated: Instead it causes a block descriptor to be allocated. The function pointer initially stored in the block descriptor is an installation function `install`. When invoked, the `install` function generates a new function `f` for the block, places a pointer to `f` in the block descriptor, and finally invokes `f`. Subsequent jumps to that block will invoke `f` directly. Figure 6.4 shows how `install` is implemented, as well as the function used to retrieve the function for an address.

Indirect jumps are also necessary to simulate self-modifying code. A simulated write to an address containing code requires that any basic blocks overlapping that address be invalidated. Notice, that multiple blocks may overlap an address if a large block has multiple entry points. Because jumps all pass through the block descriptor, invalidation is easily accomplished by replacing the function pointer in the block descriptor with `install`. Handling writes into the current block requires

```
//GENERIC INSTALL FUNCTION:
block_desc install(block_desc bd) {
 _ fe = generate_block(bd.addr);

 bd.f = fe;
 return fe(bd);
}

//BLOCK LOOKUP FUNCTION:
block_desc get_block(address a) {

 try {
   return Hashtable::lookup(code_cache,a);
 } handle y { ; }

 _ bd = new block_desc(install,a);

 Hashtable::insert(code_cache,a,bd);

 return bd;
}
```

Figure 6.4: Support functions for RTCG Simulator.

generating a new block for the current `pc`, and exiting the current function immediately. Our implementation does not handle either of these scenarios although it does check that no writes occur into code. Fortunately none of the simulated applications use RTCG!

In this description of code generation, each block is represented by a function, and simply calls the function for the next block when it is done. If each call uses the stack (they do for the return address at least), then this strategy will cause the stack space to be quickly exhausted. Fortunately, the Cyclone compiler implements tail-call elimination. Each of the function calls terminates the current function, so the Cyclone compiler is able to free the current stack frame before allocating space for the next one.

In languages, like C, where the calling convention forbids tail-call optimization in general, *trampolining*, an alternate (slightly less efficient) technique, can keep the program running in constant space. (Section 3.4.2 thoroughly describes trampolining.) Our implementation can be compiled to either rely on tail-call elimination or trampolining.

Other code-generation strategies, than the one described, are possible. For example, one could predict forward branches not taken and continue generating a larger super-block of code. Such an aggressive strategy runs the risk of generating exponentially more code than the initial program if not carefully controlled. An alternative strategy might evict infrequently used block from the code cache to limit memory use. Or one could use the simple interpreter the first time a block is accessed and only generate code after the second use. Because there are so many strategies, we have chosen to evaluate the simple straightforward one presented above. We can then use that evaluation as the basis for choosing among the rest.

6.3 Experimental Evaluation

To evaluate our implementation of the simulator, we compare the performance of the Cyclone port to the C version and the RTCG version. Our goal is to understand the factors that underly the observed performance so that we can generalize from this application to other applications.

In the following experiments, we use the programs from the SPEC CPU95 benchmark suite as inputs to the simulator. This suite was developed to test processor performance and is commonly used in conjunction with the SimpleScalar Tool Set. Therefore, the performance improvements we may attain are those of most interest to users of the system. Also, because the benchmarks were designed to test processor performance, they provide a thorough test of the simulator's capabilities.

6.3.1 Methodology

All experiments in this section were run on an unloaded 750-MHz Pentium with 256 MB of memory running Linux RedHat 7.0. Timing measurements were taken by running the simulator with various inputs and measuring the running time using `getttimeofday`. Each experiment was run at least 3 times. We always present the time for the best run.

A special version of the simulator was instrumented to measure memory allocation, the amount of code generated, and the time spent in system calls. These fine-grained timing measurements were made using the processor's cycle counter.

6.3.2 SPEC CPU95

SPEC CPU95 consists of 18 medium compute-bound applications and their inputs. Eight of the benchmarks are integer intensive and make up SPECINT95.

- *099.go* plays Go against itself repeatedly. As input the game takes an initial board position.
- *124.m88ksim* implements a simulator for the Motorola 88100 microprocessor. Like our simulator, m88ksim also takes in a binary executable and simulates it.
- *126.gcc* is version 2.5.3 of the GNU C compiler compiled to generate Sparc assembly language. As input the benchmark uses pre-processed C source files and very high-levels of optimization.
- *129.compress* applies Lempel-Ziv compression to a buffer in memory. The benchmark pre-allocates a large 14-MB buffer and takes the amount of this buffer to compress as its input.
- *130.li* is a Lisp interpreter written in C.
- *132.ijpeg* reads a PPM image file into memory, and repeatedly compresses it using different compression settings. The benchmark program comes with three different images.
- *134.perl* is a Perl interpreter. The benchmark comes with several Perl files to use as input.
- *147.vortex* is an object-oriented database. The benchmark comes with a 45MB database, and different queries to apply against the database.

The remaining ten benchmarks (SPECFP95) are floating-point intensive.

- *101.tomcatv* is a vectorized mesh generation program. Tomcatv performs finite difference approximation and LU factorization on two two-dimensional arrays.

- *102.swim* uses finite-difference approximations on a two-dimensional grid to solve a system of shallow water equations.
- *103.su2cor* computes the masses of elementary particles using a monte carlo method.
- *104.hydro2d* uses the hydrodynamic Navier-Stokes equations to compute galactical jets.
- *107.mgrid* computes a three-dimensional potential field using a multigrid solver. The input specifies a grid size, a number of points to solve, and a number of timesteps.
- *110.applu* solves five coupled partial differential equations. Applu is also used in the NAS benchmark suite [BBB+91]. The purpose of the equations is to model a computational fluid dynamics system on a three-dimensional grid.
- *125.turb3d* solves the Navier-Stokes equations using a pseudo-spectral method for the purpose of simulating turbulence in a cube with periodic boundary conditions in each dimension.
- *141.apsi* is an atmospheric simulator designed to compute variations of potential temperature, wind components, mesoscale vertical velocity pressure, and distribution of pollutants in a three-dimensional environment.
- *145.fpppp* simulates the two electron integral derivative, an important kernel for simulating quantum chemical reactions. The input is the number of atoms to simulate. The execution time is proportional to the fourth power of the number of atoms.
- *146.wave5* solves Maxwell's equations in a two-dimensional mesh. The computation is used to study plasma phenomena.

(In the future we will drop the leading numbers in benchmark names.) Because CPU95 takes 48 hours to run on a 40MHz machine, and `sim-fast` runs at 15MIPS on our test machine (a 750MHz Pentium), using the full inputs was impractical. Instead we use an input set favored by other users of the simulator, namely that selected by Doug Burger in his thesis [Bur98]. The CPU95 suite comes with three data sets for each benchmark: a training set (**train**) for driving profile-driven optimization, a test set (**test**) for establishing correctness, and the full reference input (**ref**) for official runs. Most of the inputs we use are either taken from one of these sets or use a reduced number of iterations of the data from **ref**. To justify these choices, Doug Burger measured the performance of each benchmark and showed that the reduced input had a similar instruction mix to the reference set, and spent less than 20% of its time in initialization. Table 6.1 shows the input set used for each benchmark.

Table 6.1: Simulator experimental inputs

SPECINT95	Input	SPECFP95	Input
go	test	tomcatv	ref (60 iterations)
m88ksim	train	swim	ref (45 iterations)
gcc	ref (largest file only)	su2cor	ref (62 iterations)
compress	400KB	hydro2d	ref (6 iterations)
li	train	mgrid	test (4 timesteps)
ijpeg	train	applu	ref (5 iterations)
perl	train	turb3d	ref (2 iterations)
vortex	ref (1 iteration)	apsi	ref (6 iterations)
		fpppp	train
		wave5	ref (10 iterations)

Table 6.2: Instruction Mix of CPU95 benchmarks

		percentage					
	instructions(1e6)	int	mult	fp	mem	ctrl	sys
go	16390	56	0	0	29	15	0.000000
m88ksim	119	48	1	0	31	19	0.000039
gcc	1441	39	1	0	41	20	0.000052
compress	1238	45	2	0	32	20	0.000001
li	183	35	0	0	42	23	0.000020
ijpeg	1463	59	7	0	25	8	0.000039
perl	40	34	1	0	46	19	0.000225
vortex	7713	31	0	0	53	17	0.000139
tomcatv	10652	55	0	12	27	6	0.000026
swim	2846	37	0	28	32	3	0.000001
su2cor	11549	41	1	17	33	8	0.000038
hydro2d	2443	40	0	22	25	12	0.000017
mgrid	480	35	1	26	36	2	0.000012
applu	2040	47	1	23	25	3	0.000002
turb3d	2837	55	3	14	23	5	0.000002
apsi	318	39	4	21	31	5	0.000043
fpppp	331	12	0	33	53	1	0.000010
wave5	13073	41	1	20	33	6	0.000000

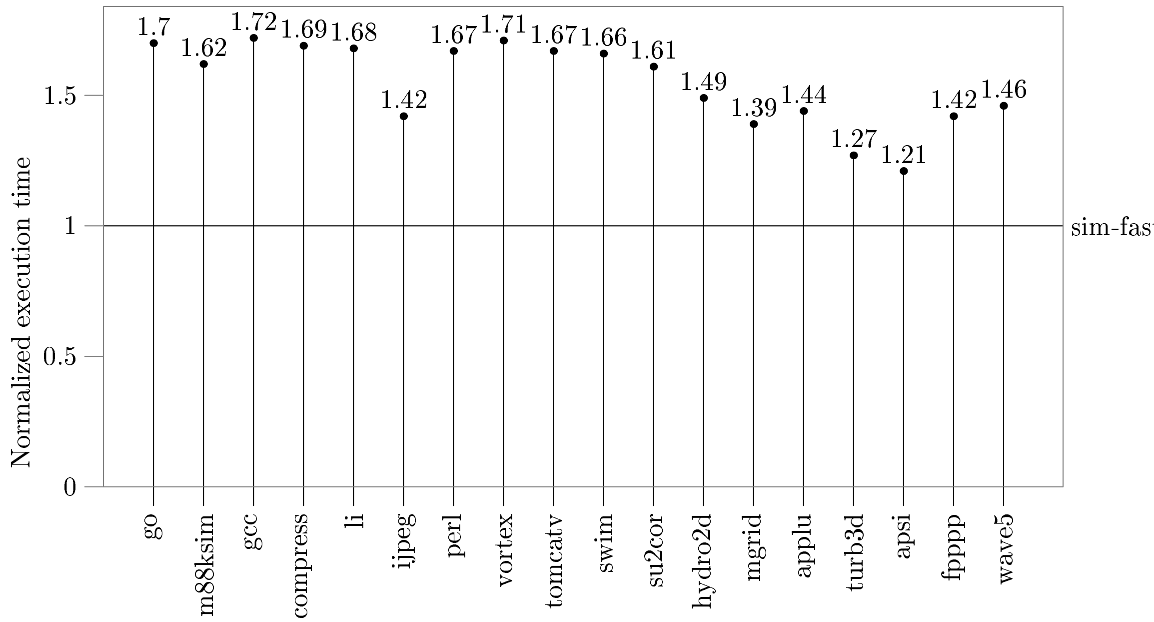


Figure 6.5: Comparing C and Cyclone versions of the simulator

Table 6.2 shows the total number of instructions simulated for one run of our standard inputs, and a breakdown of the dynamically executed instructions. Instructions are divided into classes: integer instructions except multiplication (*int*), multiplication instructions (*mult*), memory operations (*mem*), floating-point instructions (*fp*), and control instructions (*ctrl*). Multiplication instructions are distinguished because, unlike the other arithmetic instructions, they are implemented via a function call. These numbers were obtained by instrumenting the simulator.

6.3.3 Comparison to C

Figure 6.5 compares the performance of the plain Cyclone version to the C version `sim-fast`. For SPEC95, the Cyclone version is always between 1.2 and 1.75 times slower than the C version. We can try to identify the potential causes of this difference by comparing the two implementations. The most obvious differences are memory management, system calls, array-bounds checks, and compiler optimizations. Surprisingly, although these differences are responsible for some of the slowdown, these are second-order effects. The largest part of the performance difference is caused by two low-level implementation details: jump-tables and flattened top-level arrays.

Because the Cyclone version uses a garbage collector and the C version does not, memory management is a significant difference between these two implementations. As shown in Table 6.3, garbage collection consumes less than .1%. This result is not surprising because the simulator does not generate any garbage (simulated

Table 6.3: Breakdown of run time (non-rtcg version)

		percentage		
	time(seconds)	gc	system	bounds check
go	1634	0.00	0.00	-0.35
m88ksim	12	0.02	0.05	2.23
gcc	152	0.01	0.03	2.60
compress	128	0.02	0.00	2.43
li	19	0.01	0.01	2.43
ijpeg	171	0.01	0.01	3.45
perl	262	0.00	0.00	3.24
vortex	813	0.01	0.07	3.53
tomcatv	1075	0.01	0.06	0.59
swim	281	0.00	0.00	3.06
su2cor	1206	0.01	0.09	1.77
hydro2d	246	0.01	0.04	1.77
mgrid	57	0.01	0.00	0.83
applu	225	0.01	0.00	2.87
turb3d	407	0.01	0.00	1.62
apsi	56	0.02	0.01	1.98
fpppp	37	0.00	0.00	4.83
wave5	1447	0.00	0.00	2.43

pages are never freed). Furthermore, most of the allocated memory (the simulated pages) contains no pointers and need not be scanned.

In the same table, system calls are shown to consume less than .1% of execution time. (The system call times also include any allocation or collection costs they incurred.) The Cyclone version uses C wrapper functions that copy buffers to access the underlying system routines. Because the CPU95 benchmarks were chosen to be CPU bound they do not make many system calls; therefore the overhead associated with copying buffers does not significantly affect performance.

A surprising result is that array-bounds checks are not significant. Table 6.3 shows the savings obtained by turning off bounds checks. In every case the savings is less than 5% of run time. This fact is surprising because every simulated register access requires a bounds check. Eliminating the check eliminates two instructions of the four needed for an array access. We conjecture that the savings are small for two reasons: (Amdahl's law) compared to the loop's fixed costs and the cost of decoding the register numbers, the cost of a bounds check is low; and (pipelining) the processor is able to keep the bounds-check instructions in the instruction pipeline making them essentially free.

Examining the assembly generated for the inner loop by the C compiler reveals two relevant differences. The first is that the C version stores the register arrays in a global variable, whereas the Cyclone version uses a local variable. By using a global variable, the C version eliminates the load from the stack; thus saving one memory access per register access. Whereas eliminating two non-memory instructions made no meaningful difference, removing one memory instruction might significantly reduce memory traffic. Unfortunately, global variables in Cyclone are implemented with an extra level of indirection to allow them to be assigned. Therefore, using a Cyclone global variable would not accrue the same advantage.

The second significant difference is that the C version implements the main loop's switch with a jump table. Cyclone instead uses a jump tree to implement switches. The SimpleScalar architecture has 118 instructions. The jump tree takes on average 14 instructions to reach the case for any given instruction. The worst-case behavior is 26 instructions. The C version on the other hand takes a constant 3 instructions.

6.3.4 Effect of RTCG

Figure 6.6 compares the performance of the two Cyclone versions: the plain version and the RTCG version. A surprising feature of this graph is that some of the RTCG benchmarks are significantly slower than their plain counterparts – 3.5 times slower in one case. There are a number of possible explanations for this behavior: 1) increased garbage collection cost, 2) the cost of generating code, and 3) the instruction cache.

Table 6.3.4 shows the percentage of run time devoted to garbage collection and code generation. In every case this number is less than 2% of execution time. Therefore, neither of these effects is responsible for the observed slowdown.

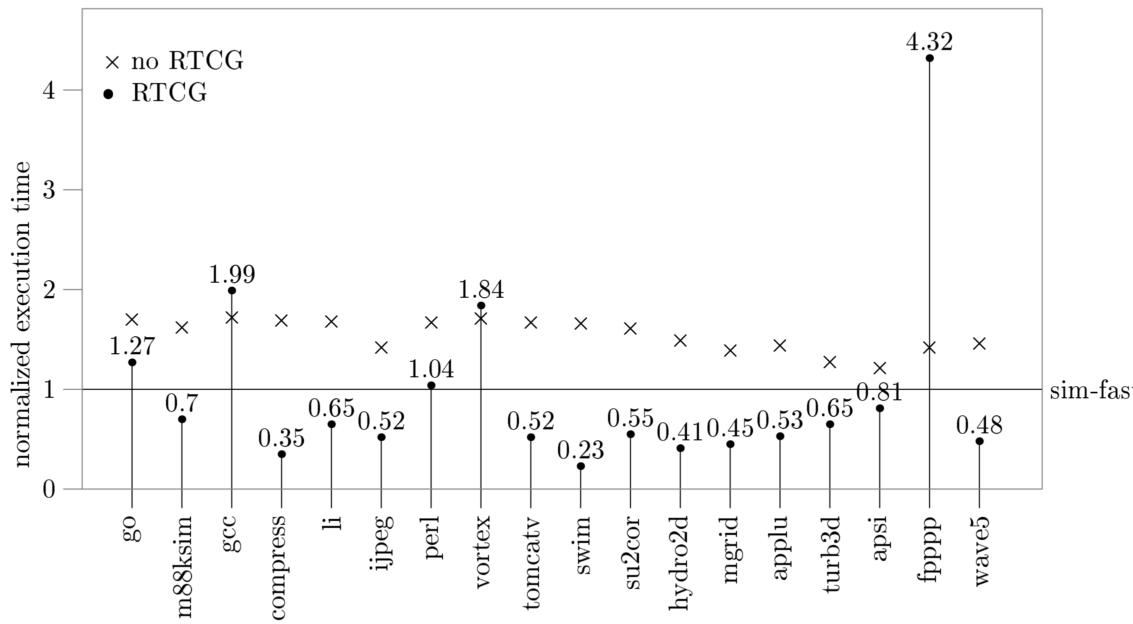


Figure 6.6: Effect on the simulator of adding RTCG.

Table 6.4: Percentage of simulator run time spent in code generation, system calls, and code generation (RTCG version).

	code generation	system call	gc
applu	0.1	0.0	0.1
apsi	0.2	0.0	0.4
gcc	0.9	0.0	1.3
compress	0.1	0.0	0.0
fpppp	0.1	0.0	0.2
go	0.0	0.0	0.1
hydro2d	0.1	0.2	0.1
ijpeg	0.1	0.0	0.1
li	0.2	0.0	0.4
m88ksim	0.4	0.1	1.0
mgrid	0.1	0.0	0.2
perl	0.0	0.0	0.1
su2cor	0.1	0.4	0.0
swim	0.1	0.0	0.1
tomcatv	0.1	0.3	0.0
turb3d	0.0	0.0	0.0
vortex	0.2	0.2	0.1
wave5	0.0	0.0	0.0

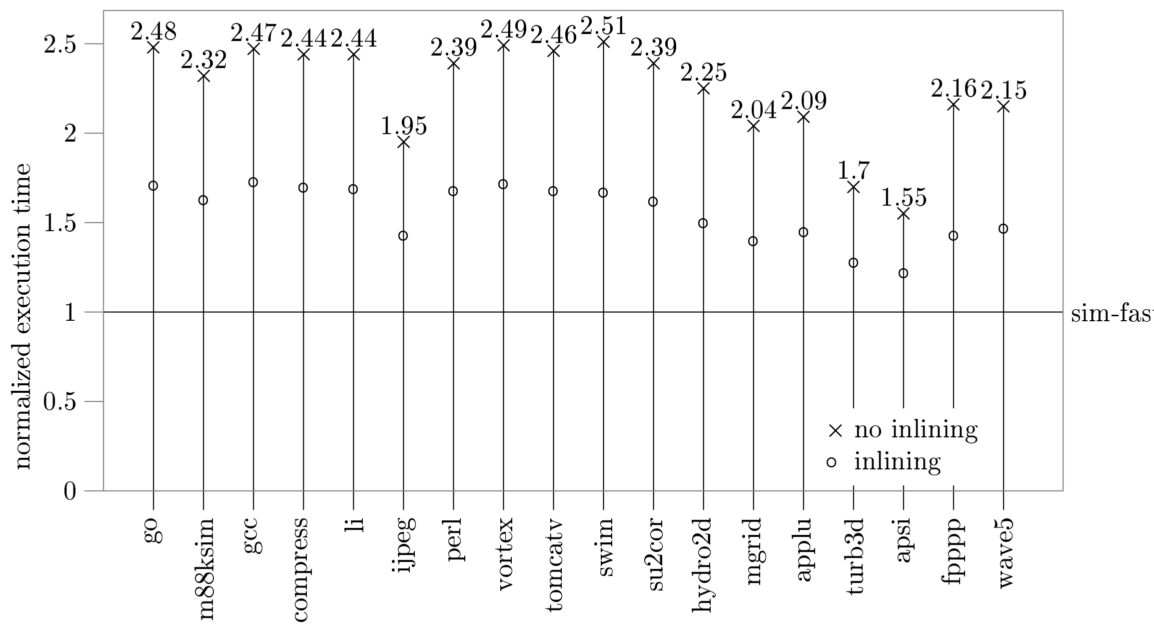


Figure 6.7: Effect of inlining memory accesses on the Cyclone version (no RTCG).

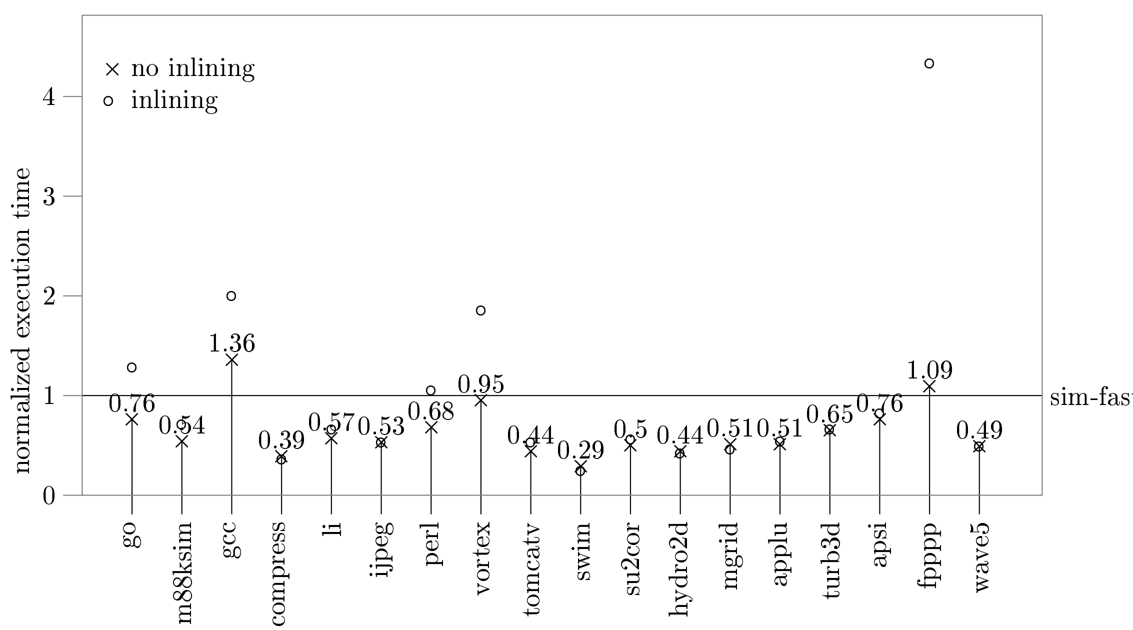


Figure 6.8: Effect of inlining memory accesses (RTCG).

Table 6.5: Size of generated code to simulated code (in KB).

	code size	RTCG size	ratio	RTCG inlined size	ratio
go	571	2821	4.9	5222	9.2
m88ksim	92	541	5.9	1051	11.4
gcc	1082	6924	6.4	12275	11.3
compress	29	177	6.1	344	11.9
li	52	332	6.4	696	13.3
ijpeg	137	769	5.6	1644	12.0
perl	128	814	6.4	1568	12.2
vortex	489	2872	5.9	6775	13.9
tomcatv	75	419	5.6	828	11.0
swim	79	433	5.5	852	10.8
su2cor	185	877	4.7	1875	10.1
hydro2d	140	789	5.6	1551	11.1
mgrid	81	454	5.6	920	11.4
applu	183	900	4.9	2111	11.5
turb3d	135	732	5.4	1638	12.1
apsi	214	1126	5.3	2787	13.0
fpppp	183	961	5.2	2885	15.7
wave5	170	899	5.3	2053	12.1

To test the possibility that the instruction cache might be the cause, we undo an optimization used by `sim-fast`. In the interpreter `sim-fast` obtained a significant performance improvement by inlining the function that simulates memory reads and writes (as shown in Figure 6.7). One result of inlining is that the generated code is significantly larger. Table 6.5 shows the effect of inlining memory accesses on the size of the generated code. For the benchmarks most severely affected by inlining; fpppp, vortex, and gcc; the size of the generated code doubles and in the case of fpppp triples.

In Figure 6.8 we present the performance obtained by disabling inlining for the RTCG code only. In almost all cases the result is a significant improvement. One consequence of this observation is that one cannot use small benchmarks to predict the performance of large programs. Large programs, as demonstrated by `fpppp`, will exhaust the instruction cache.

Since the instruction cache plays such a prominent role in determining performance, reducing the size of the generated code is important. Table 6.5 shows that the generated code is between 4.9 and 6.4 times larger than the code it was generated from. Therefore, we expect performance degradation if this factor is enough to make the key loops of a program larger than the instruction cache. This blow-up factor could be reduced if TAL/T supported holes in arbitrary instructions. The template for the ADD instruction (shown in Figure 6.3) could be reduced

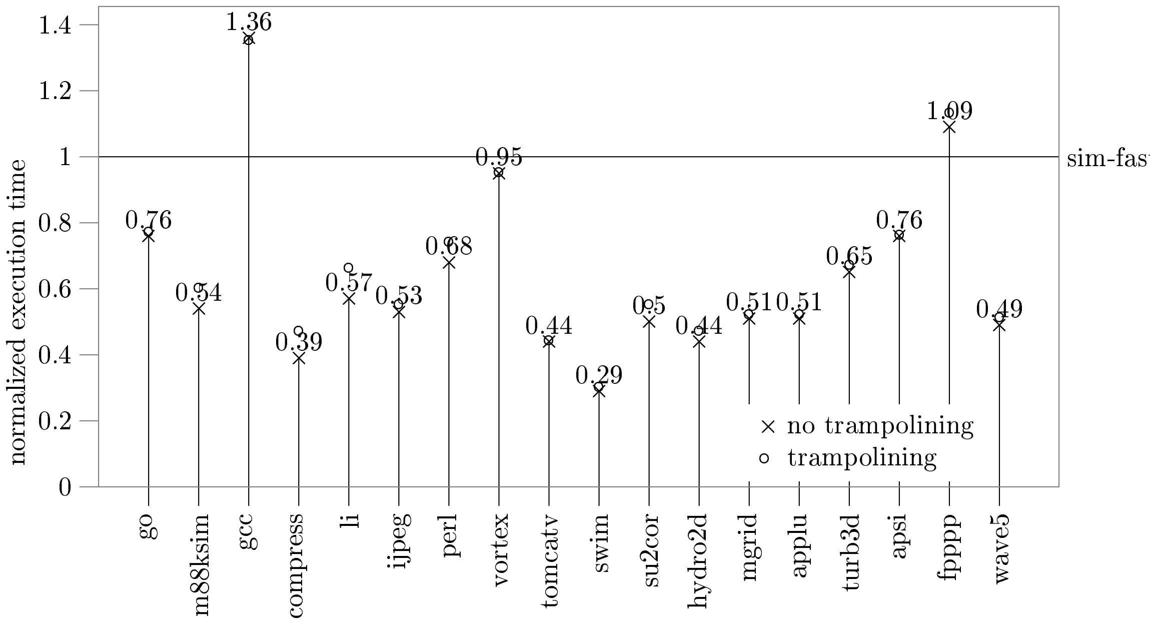


Figure 6.9: Performance comparison of trampolining to tail-call elimination

significantly.

Trampolining introduces additional overhead for each jump instruction. Tail-call elimination should yield better performance. Figure 6.9 compares the performance of two RTCG versions: one using trampolining and the other relying on tail-calls. The results show that tail-calls only provide a small benefit over trampolining.

Although RTCG has provided significant speedups, could we obtain similar speedups by other means? The savings obtained by RTCG stem from eliminating the fetch and decode overhead for each instruction. The equivalent optimization without RTCG would extract the register numbers in advance and perhaps use one function per kind of instruction. We speculate that this approach would not be as effective as RTCG because the register numbers would still have to be loaded and the `pc` register incremented. By using RTCG, the `pc` is encoded in the native program counter, and register numbers occur as constants in the generated code. In the future, we would like to explore the difference between these two approaches.

Figure 6.10 compares the best RTCG implementation (that does not inline simulated memory accesses) to the best non-RTCG implementation (that does inline simulated memory accesses). In all cases, RTCG provides a benefit: On average, the RTCG Cyclone version is 1.8 times faster than `sim-fast`. Our experiments suggest that there are two main factors that limit this benefit: 1) the instruction cache, and 2) restrictions on where holes can occur. The larger lesson here is that RTCG systems must focus on the size of generated code as well as its speed. Small increases in code size can lead to large increases in the running time.

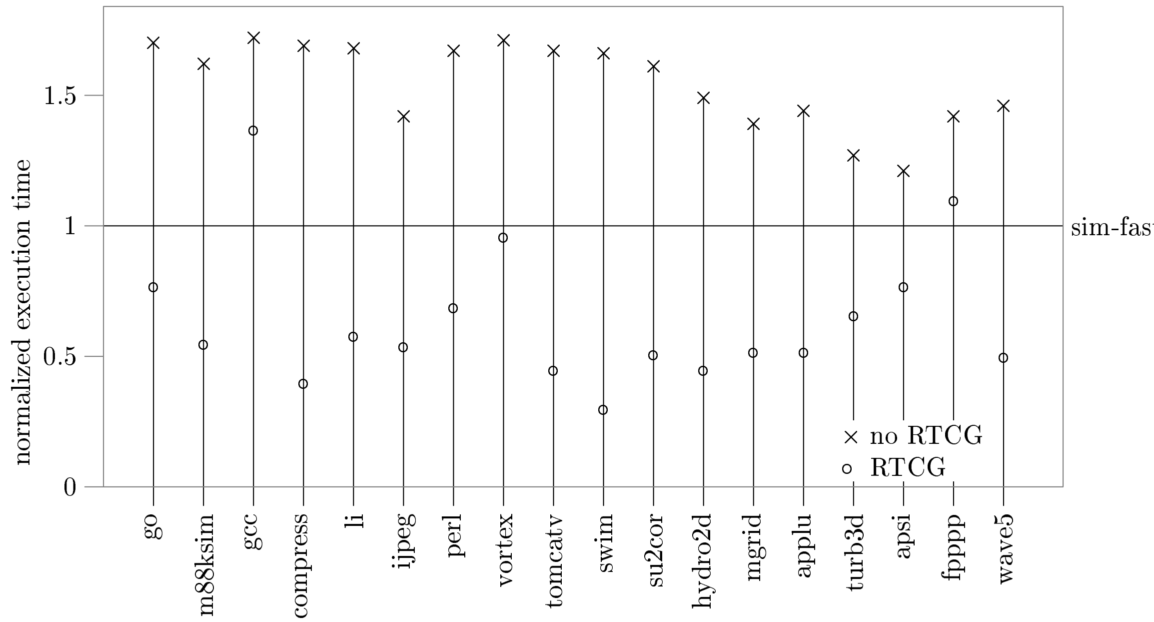


Figure 6.10: Comparison of best RTCG version to best standard version.

The success of RTCG does not depend on finding applications where specialized functions will be sufficiently used. Rather, it hinges on finding applications where generating specialized functions does not nullify the instruction cache.

Chapter 7

Compiling Cyclone

The Cyclone compiler translates Cyclone into an intermediate representation with explicit support for templates before emitting machine instructions. This internal representation permits optimization of dynamically generated code. In our compiler, after optimization and register allocation, a one pass translation produces assembly language. This chapter defines a subset of Cyclone (Mini-Cyclone), the Cyclone Intermediate Representation (CIR), and presents a formal translation from Mini-Cyclone into CIR.

In the next chapter we show how to perform standard compiler optimizations over CIR.

7.1 Mini-Cyclone

For the purpose of describing the compilation of Cyclone, it is convenient to formally define it. This is impractical for the complete language. Instead we define Mini-Cyclone – a subset of Cyclone capturing the aspects most relevant to RTCG.

Figure 7.1 presents the abstract syntax for Mini-Cyclone. The primitive values are integers (i) and function names (f). The other standard expressions are variables, addition, and function application. Similar to Cyclone, there are two expression forms for RTCG: `codegen` d generates a function and returns a pointer to it. `fill` e occurs in a generated function. Its meaning is to evaluate e as code in the function doing the code generation and embed the result, as a constant, in the generated function.

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

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

Figure 7.1: Mini-Cyclone Syntax

the computed code is simply s. Otherwise, each nested `cut` statement suspends generation of code from the `splice` statement, executes its argument, and then resumes generating code from the `splice` statement.

A Mini-Cyclone function takes a fixed number of arguments and executes its body, which is just a statement. For simplicity here, we ignore types and assume that code is well-formed. Compiling Cyclone types into CIR types is also challenging but is largely similar to the challenge faced by certifying compilers that do not support RTCG [MCG+99, GM00, TMC+96, Mor95]. We focus solely on compiling the term language.

7.2 The Cyclone Intermediate Representation (CIR)

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 chapter, 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.

CIR will have explicit support for templates. Before continuing, recall that a template is a sequence of machine instructions with distinguished locations called holes. To generate code, templates are successively copied into a code region, and the values pointed to by holes are replaced. The generating program controls which templates are copied and their order, as well as the values used to *fill* holes. There are two kinds of holes. *Value holes* contain constants such as integers and pointers to data structures and are used to control the values manipulated by the generated code. *Jump holes* occur in jump instructions and allow the generating program to change the control-flow of the generating program. Jump holes, unlike value holes, are necessary for generating correct code as relative jumps are by definition dependent on the way in which the templates are copied.

Figure 7.2 shows the syntax for CIR. A CIR program is a collection of func-

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

Figure 7.2: CIR Syntax

tions. 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, otherwise it jumps to d_2. The destination $\circ$ is a "fall through" — in block B_i of a function, $\circ$ is equivalent to the label of B_{i+1}.

A CIR function represents either code that is directly executable or a *container* for templates. Executable functions must not contain holes, whereas containers can. The template names on the blocks within a container are used to group blocks into templates. Specifically, a template t is the list of blocks with template name t.[1] Executable functions use containers to generate code dynamically by *copying templates* (t) into *code regions* (r) and *filling* holes ($[h]$). We call a copy of a template an *instance* (p). Holes ($[h]$) allow generating code to specialize instances. Generating code can dictate the control flow of generated code by filling destination holes, and can insert values into the generated instructions by filling value holes

[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.

$(x := [h])$.

The instruction $r :=$ start l allocates a code region and binds it to r. All templates copied into r should come from the container function named l. The instruction $p :=$ copy t into r puts an instance of t after the instances previously copied into r.[2] The term p becomes a reference to the new instance. The two fill instructions (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 create 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 7.3 shows the `dot_gen` example from Section 3.2 in CIR (extended with arrays, etc.). The translation of Section 7.3 would introduce more labels and variables but would otherwise produce similar output. Notice that the translation of *dot* contains holes and multiple templates, so it is not executable. When *dot_gen* is called, it creates a code region r in which to generate a function. It then copies *dot*'s prologue, t_1, and enters the loop consisting of blocks l_3 and l_4. The loop copies an instance of t_2 and binds p_2 to it. p_2 is used when filling holes h_1 and h_2. Its purpose is to specify which instance's holes to fill. Hole h_1 is filled with the value in $u[i]$ and hole h_2 is filled with the counter i. After the loop, *dot_gen* copies t_3 and ends the code generation.

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

7.3 Compiling Mini-Cyclone

This section describes a translation from Mini-Cyclone (Section 7.1) to CIR (Section 7.2), beginning with some highlights.

Translating a lexically scoped language with RTCG into an unstructured block-

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

$$\begin{array}{lll}
(u) & (dot_gen, _, & r := \underline{\texttt{start}}\ dot \\
 & & p_1 := \underline{\texttt{copy}}\ t_1\ \underline{\texttt{into}}\ r \\
 & & i := 0, \\
 & & \underline{\texttt{jmp}}\ \circ) \\
 & (l_3, _, & tst := i < size(u), \\
 & & \underline{\texttt{jnz}}\ tst\,?\,l_5 : \circ) \\
 & (l_4, _, & p_2 := \underline{\texttt{copy}}\ t_2\ \underline{\texttt{into}}\ r \\
 & & \underline{\texttt{fill}}\ p_2.[h_1]\ \underline{\texttt{with}}\ u[i] \\
 & & \underline{\texttt{fill}}\ p_2.[h_2]\ \underline{\texttt{with}}\ i \\
 & & i := i + 1, \\
 & & \underline{\texttt{jmp}}\ l_3) \\
 & (l_5, _, & p_3 := \underline{\texttt{copy}}\ t_3\ \underline{\texttt{into}}\ r \\
 & & f := \underline{\texttt{end}}\ r, \\
 & & \underline{\texttt{retn}}\ f)
\end{array}$$

$$\begin{array}{lll}
(v) & (dot, t_1, & result := 0, \\
 & & \underline{\texttt{jmp}}\ \circ) \\
 & (l_1, t_2, & x := [h_1] \\
 & & i := [h_2] \\
 & & tmp := x * v[i] \\
 & & result := result + tmp, \\
 & & \underline{\texttt{jmp}}\ \circ) \\
 & (l_2, t_3, & ,\underline{\texttt{retn}}\ result)
\end{array}$$

Figure 7.3: Translation of the `dot_gen` example (simplified)

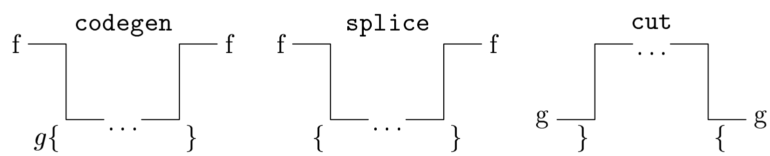


Figure 7.4: Intuition for transitions from parent to child for RTCG constructs.

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

Because Cyclone allows generated functions to generate new functions, the translation maintains an ordered sequence of environments. The environment for function f directly precedes that of g exactly when f generates g. In this case, we call f the *parent* of g and g the *child* of f. During the translation, exactly one function is *active*; instructions are emitted into the *active* function. Environments for new functions are added by `codegen`. The RTCG constructs `splice` and `cut` move from parent to child and from child to parent respectively in the existing sequence of environments. Figure 7.4 shows how the active function changes for each of these constructs. Note that the only difference between `splice` and `codegen` is that `codegen` introduces a new function. The `fill` e construct causes code to be emitted in both the active function and its parent. In the active function, the translation emits a value hole, and in the parent, it emits the code for e followed by a $\underline{\texttt{fill}}$ instruction with the variable carrying the result of computing e.

Mini-Cyclone `fill` instructions correspond to value holes.

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

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

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

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

The above difficulties make translating a language with RTCG into an unstructured block-based language inherently more complex than standard translations. To explore all the subtleties, we present a full translation from Mini-Cyclone into CIR. In order to emphasize the interesting aspects, some parts are informal; a more formal translation is given in the appendix.

7.3.1 Translation Environment and Preliminaries

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

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

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

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

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

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

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

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

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

We assume that the base syntactic classes are isomorphic and that there are functions witnessing these isomorphisms: Given an element of any class (call it a), we can generate the corresponding variable ($\mathcal{V}(a)$), label ($\mathcal{L}(a)$), hole ($\mathcal{H}(a)$), template name ($\mathcal{T}(a)$), code region ($\mathcal{R}(a)$), or instance ($\mathcal{P}(a)$). This technical trick helps us exploit subtle invariants. For example, the translation has the property that a template's instances are manipulated by the parent one at a time, so we use the template name (t) to induce an instance name ($\mathcal{P}(t)$).

7.3.2 Translation without RTCG

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

The translation of a function, $[[D]]$, takes an environment, Φ, as an argument. For top-level functions, Φ should be the empty sequence of function environments. For a D declared in `codegen`, $\Phi.$**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 $\Phi.$**active** is a new function environment. Roughly, *newFun* creates a function environment for $f(x_1, \ldots, x_n)\ s$ in which there is one empty block with label $\mathcal{L}(f)$ and template name $\mathcal{T}(f)$. Using an informal notation:

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

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

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

Expressions:

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

Statements:

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

Figure 7.5: Translation of simple statements and expressions into CIR.

For certain expressions and statements the translation is straightforward. Figure 7.5 shows the translation for these easy cases. Most of the work lies in generating fresh variables to hold intermediate results. The term *emit* Φ ι updates Φ such that ι is appended to the end of the active function's current block. Analogously, *emit* Φ χ replaces the control transfer of the current block with χ. Finally, $e_1; e_2$ means, "Do e_1 then e_2."

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

Figure 7.6 shows the translation for `if` and `while`. The term *changeBlock* puts a new block in the active function and makes the new block the current one. *changeBlock* returns a pair of the old current block's label and the new current block's label. The new block has the same template name as the old block. The term *emit* Φ b: χ replaces the transfer in block b with χ.

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

Figure 7.6: Translation of control-flow statements into CIR.

7.3.3 Translation with RTCG

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

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

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

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

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

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

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

Figure 7.7: Translation of `cut` into CIR.

So *child*(Φ) makes the child active and *parent*(Φ) makes the parent active. Translation of well-formed source code never applies *head* or *tail* to an empty list.

The translations for `cut` and `splice` shown in Figure 7.7 are pleasingly symmetric. (The helper function *newTemplate* is described below.)

To understand the way the context is manipulated in Figure 7.7 recall that the purpose of `cut` s is to execute s in the parent of the function in which the `cut` appears. Dually, the argument to `splice` is part of the child function. In both cases, we have a child-to-parent transition (beginning of the cut, end of the splice) and a parent-to-child transition (end of the cut, beginning of the splice). For child-to-parent transitions, we end the current block so that control will flow to a new template that an ensuing parent-to-child transition creates. For parent-to-child transitions, we add a new template to the child because the parent may choose to copy the code after the transition at different times than the code before the transition. Figure 7.4 depicts these transitions.

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

The translation of `fill` e is straightforward. In the parent, we translate e and emit a <u>`fill`</u> instruction for a new hole. In the child, we emit a value hole. Figure 7.8 shows the translation. (*currentTemplate* retrieves the current block's template name.)

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

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

Figure 7.8: Translation of $\underline{\texttt{fill}}$ into CIR.

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

Figure 7.9: Complete definition of genDest.

return the destination's label. If different, we create a hole, emit a fill for the hole in the parent, and return the hole. Figure 7.9 shows the definition of genDest.

7.4 Implementation Details

An underlying run-time infrastructure provides the support for templates required by CIR. Although a template is simply a fragment of machine code, code regions are dynamically sized and may contain pointers.

Because an arbitrary number of templates might be copied into a code region, there must be some mechanism for resizing it. One possibility is to have the programmer specify a buffer size and abort when that size is exceeded. This choice is impractical because it forces the programmer to restart the code generation with a larger code region. Making code generation re-entrant is challenging and expensive in an imperative language. Another alternative would allocate a new code region and link it to the existing code region when space is exhausted. In this scheme, the run-time system would insert a jump from the end of the current code region to the beginning of the new one; thus preserving the dynamic semantics. The additional jump would result in less efficient generated code. Finally, the runtime system could simply allocate a new code region twice as large as the original one and copy the existing contents into the new region. This choice imposes no penalty on the running code but requires dynamically generated code to be position independent and makes code generation slower. The Cyclone implementation uses this copying approach.

The run-time system also must ensure that pointers into the heap in code regions are traced by the garbage collector. Heap pointers are introduced into code regions by filling holes. One possibility is to have the garbage collector trace through the generated code. Because pointers in code are not aligned, the garbage collector must treat code especially. An accurate solution would require parsing the instruction stream – an expensive operation on the Intel x86 architecture. A conservative solution would scan all words in the code region indiscriminately, and lead to many false positives. An alternative solution would associate a data structure recording all the pointers with the code region. While filling holes this data structure would be updated with a copy of the hole's contents as well. This is what is done in the Cyclone implementation. The code region is an allocated block of memory whose first word is a pointer to a linked list of the values used to fill holes. Each fill operation updates this linked list.

Chapter 8

Analysis and Optimization

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

This definition of CFG does not apply to container functions because there is no control flow in the containers *per se*. Rather, we are interested in how control may flow in the functions *generated* from the container. We want the CFG for a container function to safely approximate control flow for all functions that may eventually be generated from it. We therefore generalize the definition of CFG as follows: A CFG for a container function F is a directed graph, with inter-template and intra-template edges, whose nodes are the labels in F and where control may flow from an *instance* of l_1 to an *instance* of l_2 only if there is an edge of some kind from l_1 to l_2. The two kinds of edges represent different degrees of precision:

- An inter-template edge between (l_1, l_2) allows control to flow from any instance of l_1 to any instance of l_2.

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

8.1 Examples

To understand our decision to distinguish intra-template and inter-template edges, consider the two (slightly embellished) Mini-Cyclone functions in Figure 8.1. The behaviors of f and g are not as important as the similarity of their (simplified) translations shown in Figure 8.2.

The significant difference between these two container functions is that the translation of the loop in f has an intra-template back-edge (from f_2 to f_1) whereas g has a cycle caused by an inter-template edge (from g_3 to g_1). Despite this difference, the control-flow graphs (see Figure 8.3) look quite similar if we ignore

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

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

Figure 8.1: Cyclone code demonstrating the distinction between inter-template and intra-template edges.

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

Figure 8.2: Simplified translations into CIR of the container functions from Figure 8.1.

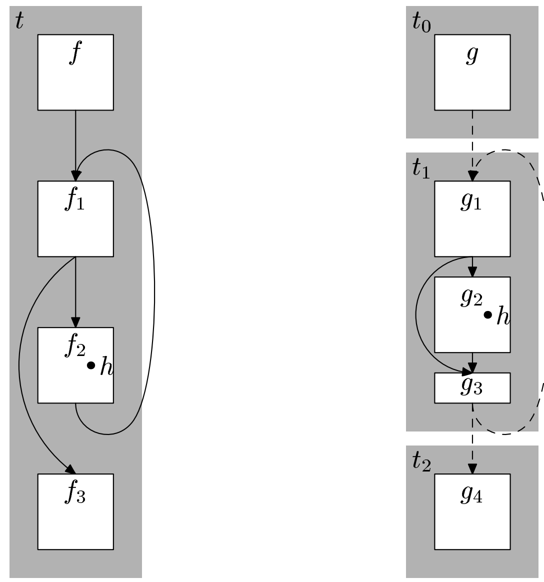


Figure 8.3: Control-flow graphs for f and g. Solid arrows denote intra-template edges; dashed arrows denote inter-template edges.

the edge kinds. Yet f and g are used very differently: Functions generated from f have one copy of each block in the order they appear in f. Functions generated from g are loop-free and contain n instances of t_1.

The CFGs capture most, but not all, of this information. The graph for f tells us that the loop is intra-template, but we do not know how many times t may be copied.[1] From the CFG for g, we must conservatively assume that the inter-template edge from g_3 to g_1 might give rise to an intra-instance edge or an inter-instance edge in a generated function. In this sense, inter-template edges represent a proper superset of the possibilities that intra-template edges represent. However, we have two ways to reason that there will not be an intra-instance edge from g_3 to g_1. First, only hole destinations can give rise to intra-instance control flow and g_2 has a fall-through destination. Second, the translation in Section 7.3 never fills a hole with a label from the same template.

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

8.2 Dataflow Analysis over CIR Functions

The reason we have distinguished inter-template edges is that holes are constant in intra-template loops, but not necessarily in inter-template loops. After all, a loop

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

containing an inter-template edge can lead to a generated function with multiple instances of a template, each with its holes filled with *different* constants.

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

On the other hand, distinguishing edges allows an analysis to determine that more expressions are loop invariant. Given a loop in a conventional CFG, we say an expression in the loop is invariant if it must have the same value from the time code in the loop is entered until code not in the loop is entered. With RTCG, we mean that in all generated functions, all instances of an expression must have the same value—and the same value as all other instances—from the time an instance of code in the loop is entered until an instance of code not in the loop is entered. Other definitions of invariance are less useful for optimization. For example, the expression $x * m$ is invariant in f_2, but not in g_2. An optimizing compiler could hoist the first two instructions from f_2 to f, but hoisting the same code from g_2 to g would not preserve meaning.[2]

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

8.3 Transformations over CIR Functions

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

The changes in a parent that intra-template transformations require are straightforward. Essentially, a hole might be deleted (e.g., dead-code elimination), duplicated (e.g., copy propagation), or moved (e.g., instruction scheduling). It is easy to find all fills of the hole in a parent and do the corresponding deletion or duplica-

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

tion. (Intra-template hole motion requires no change in a parent.) Note that the soundness of an intra-template transformation may rely on inter-template information. For example, suppose our example function f had an additional template that assigned 0 to y and then had an inter-template jump to f_2. In this case, we could not move the hole and multiplication out of the loop.

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

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

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

8.4 Control-flow Graphs for CIR functions

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

There are two distinct kinds of edges in the CFG for F, intra-template edges ($\texttt{intra}(F)$) and inter-template edges ($\texttt{inter}(F)$). The edges in $\texttt{intra}(F)$ represent control flow within a template instance. They arise from label destinations and from intra-template fall-through destinations. The edges in $\texttt{inter}(F)$ represent control flow that may be to another instance.[3] They arise from hole destinations and fall-through destinations in a template's last block. These edges approximate how the generator(s) of F might create control flow by filling holes and copying templates.

The computation of $\texttt{intra}(F)$ is standard: Let $F = (x_1, \ldots, x_n)B_1, \ldots, B_m$ and $B_i = (l_i, t_i, \ldots, \chi_i)$. Let d be a destination in χ_i. If $d = l_j$ and $t_i = t_j$, then $(l_i, l_j) \in \texttt{intra}(F)$.[4] If $d = \circ$, $i < m$, and $t_i = t_{i+1}$, then $(l_i, l_{i+1}) \in \texttt{intra}(F)$. No other edges are in $\texttt{intra}(F)$.

Computing $\texttt{inter}(F)$ is more interesting. Given only F, we could compute only a very conservative $\texttt{inter}(F)$: Any hole could be filled with any label of any

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

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

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

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

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

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

Instructions add edges by filling holes and copying templates:

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

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

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

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

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

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

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

These equations specify a forward dataflow problem that can be solved using standard techniques [ASU86, Muc97], starting with $in(B) = \emptyset$ for the first block of $parent(F)$. Note that we use the assumption that CFG for $parent(F)$ is already defined in the definition of $in(B)$. Using the dataflow solution and letting $\iota_{(l,i)}$ denote the i^{th} instruction in block l, we can define `copypred`:

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

Having defined the analysis for the subset of CIR produced by our translation, we now discuss how to generalize the CFG analysis to all of CIR. If F has multiple parents, we just add the edges that each parent requires. Similarly, if one parent uses multiple code regions, we just repeat the process for each code region. Because code regions and instance pointers are second-class constructs, we always know which code region an instruction affects.

The interesting extension is for mutually generating functions. That is, nothing in CIR prevents a cycle in the graph where edge (F, G) means F is a parent for G. Here we cannot compute the CFG for a parent before a child. The solution is to iterate the program-wide analysis in a manner just like conventional inter-procedural analysis. Note that our translation never produces mutually generating functions.

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

8.5 Performance of the Analysis

To measure the effect of the analysis, we studied its effectiveness for the micro-benchmarks described in Chapter 4. Prior to developing our analysis, we were computing a CFG during translation using *ad hoc* techniques, so we used that as a point of comparison. (Separating the translation and CFG creation simplified the compiler.)

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

Chapter 9

TAL/T Formalizaton

The Typed Assembly Language (TAL) system consists of a verifier that reads in Intel IA-32 object files annotated with types and checks that the object file is well-typed. Ill-typed object files are rejected. The soundness theorem we would like to prove is that every well-typed object file is well behaved. For example, well-behaved programs will not access memory they have not allocated; will not jump to memory locations that do not contain code; and will not read or write unaligned memory addresses.

This chapter presents an extension to TAL called TAL/T that supports template-based run-time code generation while maintaining TAL's formal properties. We have also extended the TAL implementation to support RTCG constructs.

The TAL/T formalism is expressed in terms of an abstract machine and a simplified type system for which the soundness theorem can be proved. The relationship between the abstract system we present and the implementation will remain informal. Formally proving such theorems for real systems is the subject of ongoing research [App01, AF00].

Our formal system consists of a syntax for machine states (M), an operational semantics defined as a deterministic transition relation between machine states ($M \longmapsto M'$), and typing judgements that define well-typed machine states($\vdash M$). These components are crafted to simplify the statement and proof of the soundness theorem while at the same time preserving a degree of realism.

As such, our formalism is an imperfect compromise. Greater realism usually complicates the proof. A long complicated proof is harder to check and therefore less convincing. But a proof for a system that does not accurately model the real implementation is of no interest. At each point in the design process we have tried to pick the simplest system that captured the salient aspects of the implementation.

Before formally stating the soundness theorem – well-typed machine states are well behaved – we must formally define "well-behaved". Unlike a real processor, our operational semantics is crafted so that only well-behaved transitions are possible. For example, there will be no transition from a state attempting to access memory that has not been allocated. A machine state M is *stuck* is there is no M' such that $M \longmapsto M'$.

$$\begin{array}{llll} \text{(instructions)} & \iota & ::= & \texttt{add}\ r_1, r_2, v \mid \texttt{bnz}\ r, v \mid \texttt{mov}\ r, v \mid \\ \text{(blocks)} & I & ::= & \iota; I \mid \texttt{jmp}\ v \mid \texttt{jmpr}\ v \end{array}$$

Figure 9.1: Core TAL/T instructions

Theorem 1 (Soundness) *If* $\vdash M$ *and* $M \longmapsto \ldots \longmapsto M'$ *then* M' *is not stuck.*

Our ultimate goal is to prove this Soundness Theorem. Our proof, given in Appendix C, uses a syntactic proof technique popularized by Wright and Felleisen [WF94]. This technique relies on two key lemmas from which Soundness follows: a Progress Lemma which states that well-typed machine states are not stuck ($\vdash M$ implies M is not stuck), and a Preservation Lemma which states that well-typed machine states step to well-typed machine states ($\vdash M$ and $M \longmapsto M'$ implies $\vdash M'$). These two lemmas are strictly stronger than the Soundness theorem in that they require the machine state to remain well-typed under reduction. Therefore, to use this proof technique, our formal system must satisfy this stronger notion of soundness.

The rest of this chapter describes the TAL/T language, abstract machine, and type system. All the formal definitions are collected in Appendix B. The soundness proof is given in Appendix C.

9.1 Overview of TAL

The original typed assembly language (TAL) has been proven sound [MWCG99] and has been soundly extended with stacks [MCGW98] and modules [GM99]. The core TAL that we use is based on the original TAL but has been changed either to simplify the proof or to model templates more accurately. We believe that stacks and modules could be added to this version of TAL as they were added to the previous version. This section informally describes the core TAL that we will extend with support for run-time code generaiton.

Like a conventional assembly language, the code in a TAL program consists of a set of labelled instruction sequences where the labels are addresses used for control transfers. However TAL programs are explicitly typed – there is enough type information on the instructions and operands so that syntax-directed type-checking can be done.

Figure 9.1 shows the syntax for the TAL instructions. A block is a sequence of instructions terminating in a control-transfer. In the syntax, r is used to denote a register, and v to denote either a register or an immediate word-sized value.

The `add` $r1, r2, v$ instruction adds the contents of registers r_2 and the operand v and stores the result in register r_1. By convention, results are always stored in the leftmost register. The `bnz` r, v instruction branches to the destination address represented by v if the contents of r are not 0, otherwise it continues executing the rest of the block. The `mov` r, v instruction moves the contents of v into register

r. Finally, the instructions `jmpr` v and `jmp` v perform unconditional jumps. The instruction `jmp` v jumps to the address v represents. The relative jump instruction `jmpr` v treats v as an offset from the current program counter.

This selection of instructions is chosen to be minimal and representative. A realistic language would support additional branching instructions, a complete set of arithmetic operations, tuples, dynamic allocation, arrays, and complex data types. These features have been demonstrated in previous versions of TAL and are all part of our implementation. We have omitted them here to simplify the presentation and proof.

9.1.1 The TAL Abstract Machine

To prove TAL sound we must formally define what it means to execute a TAL program. We give TAL's semantics using a deterministic rewrite relation similar to formalisms for modelling higher-level languages, such as SECD [Lan64] and CESK [Fel87]. Our abstract machine maintains but does not use type information during evaluation. An implementation that erased types prior to execution could be easily obtained by simply erasing the types in our rewrite relation. The reason we maintain types is to facilitate the proof of soundness. As explained earlier, our proof relies on the abstract machine remaining well typed under evaluation.

A TAL abstract machine state (M) has three components: a heap (H), a register file (R), and a current instruction sequence (I). The heap is a finite partial function from memory addresses to values. The register file is a function from registers to word-sized values. We write partial functions as $\{l_1 \mapsto h_1, \ldots, l_n \mapsto h_n\}$.

Our deterministic rewrite relation ($M \longmapsto M'$) maps machine states to machine states. The rewrite rules are straightforward and directly encode the informal semantics discussed earlier. For example, the rule for addition with an immediate operand is roughly

$$(H, R, \mathtt{add}\; r_1, r_2, i; I) \longmapsto (H, R[r_1 = R(r_2) + i, \mathtt{pc} = R(\mathtt{pc}) + 1], I)$$

This rule looks up the contents of r_2 with the notation $R(r_2)$ and updates the register file contents of r_1 with the sum. The meta-variable i ranges over integer literals.

This rule advances the program counter both explicitly (by incrementing it by one) and implicitly (by popping the `add` instruction off the instruction sequence). The explicit instruction sequence in the machine state is redundant (the `pc` stores the same information): We maintain it for typing reasons that will be made clear later.

To make incrementing the `pc` register in this way meaningful, TAL models the size of code blocks. The size of a code block is the sum of the sizes of the instructions it contains. Instructions in TAL have size one except as explained below.

(machine states)	M	$::=$	(H, R, I)
(heaps)	H	$::=$	$\{l_1 \mapsto h_1, \ldots, l_n \mapsto h_n\}$
(register files)	R	$::=$	$\{r_1 \mapsto w_1, \ldots, r_n \mapsto w_n\}$
(heap values)	h	$::=$	$w \mid \mathtt{code}\Gamma.I$
(word values)	w	$::=$	$i \mid *l \mid \mathtt{rel}[i] * j \mid \mathtt{S}(i)$
(operands)	v	$::=$	$r \mid w \mid []$

Figure 9.2: Syntax for machine states, register files, heap values, word values, and code blocks.

TAL supports hole operands ($[]$) to embed values in the instruction stream relative to the `pc` register. For example in the heap

$$\{1 \mapsto \mathtt{code}\ \mathtt{add}\ r_1, r_2, []; \mathtt{jmp}\ l,\ 2 \mapsto 4 \ldots\}$$

a hole operand is used to embed the value 4 in the instruction stream. At this instant in time, the example block is equivalent to $\mathtt{code}\ \mathtt{add}\ r_1, r_2, 4; \mathtt{jmp}\ l$; however the value at location 2 in the example heap may change during the program run, whereas 4 in the "equivalent" block cannot. To make room for the embedded value, instructions containing a hole have size two (instructions may not contain more than one hole).

Given this discussion, the size of an instruction ($|\iota|$) is defined as follows: $|\iota| = 1$ if ι does not contain any holes, $|\iota| = 2$ otherwise. Therefore, the size of a code block is simply its length plus the number of hole operands it contains.

Register filas are functions from register names (r) to word-sized values (w), and heaps are partial functions from addresses (l) to heap values (h). Addresses are just integer literals. We use i, j, k, l for integers but reserve l for integers intended to denote addresses.

Heap values consist of typed code blocks ($\mathtt{code}\Gamma.I$) and word-sized values (w). Word-sized values are all integers. We augment these integers with typing annotations to direct the type checker. Ordinary integers are denoted by i. Pointers are denoted by $*l$. (We will add more annotations later when their purpose can better be made clear.)

A typed code block ($\mathtt{code}\Gamma.I$) consists of a pre-condition (Γ) and a sequence of instructions (I). The pre-condition (Γ) assigns types to each register. Control cannot be transfered to this block until the registers have the types given in Γ.

Figure 9.2 summarizes the syntax for machine states, register files, heap values, word values [1], operands and, code blocks. TAL supports operands (v) that are registers (r), word values (w), and holes ($[]$). To determine the word value represented by an operand v, we use the function $S_{H,R}(v)$ defined as follows: $S_{H,R}(r) = R(r)$, $S_{H,R}(w) = w$, and $S_{H,R}([]) = H(R(\mathtt{pc}) + 1)$.

With these definitions we can describe the semantics of the instructions. The

[1] Some of these word values will first be explained in the next section.

`mov` instruction simply moves the contents of its operand into the destination register:

$$(H, R, \texttt{mov}\ r, v; I) \longmapsto (H, R[r = S_{H,R}(v)][\texttt{pc} = R(\texttt{pc}) + |\ \texttt{mov}\ r, v|], I)$$

(The notation $R[r = x]$ replaces $r \mapsto y \in R$ with $r \mapsto x$. If is undefined if r is not in the domain of R.)

The `add` instruction as described earlier performs addition:

$$\begin{aligned}&(H, R, \texttt{add}\ r_1, r_2, v; I) \longmapsto \\ &\qquad (H, R[r_1 = R(r_2) + S_{H,R}(v)][\texttt{pc} = R(\texttt{pc}) + |\ \texttt{add}\ r_1, r_2, v|], I)\end{aligned}$$

When the branch is not taken `bnz` is easily described:

$$(H, R, \texttt{bnz}\ r, v; I) \longmapsto (H, R[\texttt{pc} = R(\texttt{pc}) + |\ \texttt{bnz}\ r, v|], I)$$

when $R(r) = 0$. When the branch is taken, it behaves just like a jump instruction.

$$(H, R, \texttt{jmp}\ v; I) \longmapsto (H, R[\texttt{pc} = l], I')$$

when $S_{H,R}(v) = *(l)$, and $H(l) = \texttt{code}\Gamma.I'$. The relative jump instruction behaves in a similar manner but uses the value in `pc` to compute the destination.

$$(H, R, \texttt{jmpr}\ v) \longmapsto (H, R[\texttt{pc} = l + R(\texttt{pc})], I')$$

when $S_{H,R}(v) = *l$ and $H(l + R(\texttt{pc})) = \texttt{code}\Gamma.I'$.

Those are all the instructions. The attentive reader may wonder how the abstract machine terminates. It does not. Like a real processor, the abstract machine will run forever. In practice, programs jump to a distinguished `exit` label to relinquish control of the processor to the operating system.

Notice too that this abstract machine forbids "bad" behaviors. For example, our operational rule for `add` does not permit pointers to be added together. A program attempting to do so would therefore be stuck – no operational rule would apply. The type system we devise in the next section is designed to ensure that well-typed programs will not become stuck.

9.1.2 Overview of the TAL Type System

For each of the abstract machine's syntactic categories there is a corresponding category of types. Heaps (H) are given heap types (Ψ). Register files are given register-file types (Γ). Word values (w) are given word types (τ). Heap values (h) are given memory types (σ).

Both heap types and register-file types are partial functions. Heap types (Ψ) are partial functions from addresses (l) to heap types (σ). Register-file types (Γ) are partial functions from register (r) to word types (τ). A heap type is written as $\{l_1 \mapsto \sigma_1, \ldots, l_n \mapsto \sigma_n\}$. The notation $\Psi(l)$ retrieves the type associated with location l. If l is not in the domain of Ψ then $\Psi(l)$ is undefined. Exactly analagous notation applies for register-file types. Roughly, our type-checking rules will ensure that if $\vdash H : \Psi$ then $\vdash H(l) : \Psi(l)$.

Word types (τ) are either integers (*int*), pointers to heap types ($*\sigma$), singleton pointers ($S(i)$), or relative pointers ($rel[i] * \sigma$). The roles played by singleton pointers and relative pointers will be discussed later. Briefly, singleton pointers will be used to type-check holes in templates; and relative pointers are needed to type-check relative jumps.

Memory types (σ) are either word types (τ) or code types (Γ). For code, Γ serves as a pre-condition. Syntactically Γ is a function assigning types to each register. For example, a possible pre-condition Γ might be

$$\{r_1 \mapsto int, r_2 \mapsto *int\}$$

The type system will ensure that jumps to a block with this pre-condition only occur when r_1 contains an integer, and r_2 contains a pointer to an integer. Code blocks do not have post-conditions because they always terminate with a jump to another code block.

The soundness theorem relies crucially on the notion of well-typed machine state ($\vdash M$). A machine state is well-typed when three conditions hold.

$$\frac{\Psi \vdash H : \Psi \qquad \Psi \vdash R : \Gamma \qquad \Psi; \Gamma \vdash I}{\vdash (H, R, I)}$$

First ($\Psi \vdash H : \Psi$), under the assumption that the heap H has type Ψ we can conclude that it has type Ψ. The heap type Ψ appears to the left and right of the turnstyle so that recursive jumps can be type-checked. Second ($\Psi \vdash R : \Gamma$), the values in the registers have the types assigned to them by Γ under the assumption that the heap has type Ψ. The assumption Ψ allows the registers to refer to addresses in the heap. Third ($\Psi; \Gamma \vdash I$), the instruction sequence is well-typed in a heap with type Ψ and a register file with type Γ. Instruction sequences are type-checked by stringing together the results of type-checking individual instructions.

The judgement for checking that an individual instruction is well-typed is written as a Hoare triple: $\{\Psi; \Gamma\}\iota\{\Psi'; \Gamma'\}$. Conceptually Ψ and Γ completely describe the state before the instruction is executed and ι transforms that state into a new state $\{\Psi', \Gamma'\}$. For example, consider the type-checking rule for the `add` instruction.

$$\frac{\Gamma(r_2) = int \qquad \Psi; \Gamma \vdash v : int}{\{\Psi; \Gamma\}\ \texttt{add}\ r_1, r_2, v\{\Psi; \Gamma[r_1 = int]\}}$$

This rule checks that r_2 and v are integers. If so then after executing the `add` instruction, register r_1 will contain an integer. Therefore, the rule updates the post-condition to reflect this fact. Most of the other instruction's type-checking rules are similarly simple. We delay their presentation until later.

When type-checking a code sequence the Hoare triples from individual instructions are stringed together. The following rule for type-checking a code sequence is almost right.

$$\frac{\{\Psi;\Gamma\}\iota\{\Psi';\Gamma'\} \qquad \Psi';\Gamma' \vdash I}{\Psi;\Gamma \vdash \iota; I}\text{(almost right)}$$

It is not right because it fails to track the `pc`.

Tracking the `pc` is important for type checking holes. To type check the instruction `add` $r_1, r_2, []$ from the state $\{\Psi;\Gamma\}$ we must derive that $\Psi, \Gamma \vdash [] : \mathit{int}$ (see the rule for `add` above). Recall from the operational semantics that the value represented by $[]$ depends on the value of the `pc` register when this code is being executed: It represents the value $H(R(\texttt{pc}) + 1)$ if the heap is H and the register file is R. By analogy we would like to look up $\Psi(\Gamma(\texttt{pc}) + 1)$ to determine the type of the value at this location.

We introduce singleton values ($\texttt{S}(i)$) and types ($S(i)$) for this express purpose. A singleton type is used to type-check a single value only. Any address pointing to a heap value of type σ can be given type $*\sigma$; but only the single value $\texttt{S}(5)$ can be given the singleton type $S(5)$. The rule for type-checking singleton types is straightforward.

$$\frac{}{\Psi \vdash \texttt{S}(l) : S(l)}$$

Singleton pointer types are useful when a value must be tracked precisely. (Because addresses are integers we will use the term singleton integer and singleton pointer interchangeably.)

Now we can correct the above rule so that it precisely tracks the value in the `pc` register using singleton pointer types.

$$\frac{\{\Psi;\Gamma\}\iota\{\Psi';\Gamma'\} \qquad \Gamma(\texttt{pc}) = S(l) \qquad \Psi';\Gamma'[\texttt{pc} = S(l + |\iota|)] \vdash I}{\Psi;\Gamma \vdash \iota; I}\text{(correct)}$$

The rule for type-checking holes should also be apparent now.

$$\frac{\Gamma(\texttt{pc}) = S(l) \qquad \Psi(l+1) = \tau}{\Psi;\Gamma \vdash [] : \tau}$$

Finally, we must describe how to type-check the control-transfer instructions. Consider first a simple jump instruction. A jump is well-typed when the precondition of the destination is a subtype of the current register-file type. (We will not formally specify the subtype relation here; it is roughly the same as width subtyping for records.)

$$\frac{\Psi;\Gamma \vdash v : *\Gamma' \qquad \vdash \Gamma \leq \Gamma'}{\Psi;\Gamma \vdash \texttt{jmp}\ v}\text{(almost right)}$$

This rule is not right because the value of $\Gamma(\texttt{pc})$ need not be related to $\Gamma'(\texttt{pc})$ – the jump instruction itself will update the value in `pc`. Indeed, if Γ' mentions the singleton pointer type of `pc` than the contents of v must be a pointer to $\Gamma'(\texttt{pc})$. This choice severely limits the kinds of programs that can be written.

Consider the following example block representing the "function" add.

$$\begin{array}{l}\texttt{code}\{\texttt{r}_1 \mapsto \mathit{int}, \texttt{r}_2 \mapsto \mathit{int}, \texttt{r}_3 \mapsto *\{\texttt{r}_1 \mapsto \mathit{int}\}, \texttt{pc} \mapsto S(5)\}.\\ \texttt{add}\ \texttt{r}_1, \texttt{r}_1, \texttt{r}_2;\\ \texttt{jmp}\ \texttt{r}_3\end{array}$$

The type of register $\texttt{r}_3$ says that it must contain a pointer to a code block expecting an integer in $\texttt{r}_1$. This pointer is serving as a return address. For multiple return addresses to be possible, the type in $\texttt{r}_3$ cannot specify a single singleton pointer type for the `pc` register.

We solve this problem in two parts: 1) a modification to the rule for type-checking pointers, and 2) a modification to the rule for type-checking jump instructions. We use the function $\sigma@l$ to eliminate `pc` from a pre-condition.

$$\sigma@l = \begin{cases} \Gamma\backslash\texttt{pc} & \text{if } \sigma = \Gamma \text{ and } \Gamma(\texttt{pc}) = S(l) \\ \sigma & \text{otherwise} \end{cases}$$

This functions checks that the pre-condition for `pc` is consistent with the location of the code; in effect specializing the type to its location. It is applied whenever a pointer is created.

$$\frac{\Psi(l)@l = \sigma}{\Psi \vdash *l : *\sigma}$$

Because every code sequence requires `pc` to be a singleton pointer type, only the above rule can create pointers to code with no pre-condition on `pc`. Therefore we can correct the type-checking rule for `jmp` to explicitly exclude the value of `pc` from consideration.

$$\frac{\Psi; \Gamma \vdash v : *\Gamma' \qquad \vdash \Gamma\backslash\texttt{pc} \leq \Gamma'}{\Psi; \Gamma \vdash \texttt{jmp}\ v}(\text{correct})$$

If `pc` is in the domain of Γ' then the subtyping relationship will fail to hold[2].

Relative jumps are the only remaining instructions. What type should v in $\texttt{jmpr}\ v$ have? It should be a pointer relative to the `pc`. Intuitively, $R(\texttt{pc}) + S_{H,R}(v)$ should be a pointer to code. We generalize pointers $*\sigma$ for this purpose.

The relative-pointer value $\texttt{rel}[i] * l$ has relative-pointer type $\mathit{rel}[i] * \sigma$ if the address $i + l$ contains a value of type σ. In the concrete value $\texttt{rel}[i] * l$, i is part of the typing annotation and not part of the value. The type-checking rule for relative-pointer types is simple.

$$\frac{\Psi(i+l)@(i+l) = \sigma}{\Psi \vdash \texttt{rel}[i] * l : \mathit{rel}[i] * \sigma}$$

[2] A number of other solutions are possible to the `pc` problem. This one extends naturally to templates.

Notice that when $i = 0$ this is the same rule we had for pointers $*l$. In the future we consider $*\sigma$ and $*l$ shorthand for $rel[0] * \sigma$ and $\texttt{rel}[0] * l$ respectively.

With these new types, the type-checking rule for relative jumps is easily expressed.

$$\frac{\Gamma(\texttt{pc}) = S(l) \qquad \Psi; \Gamma \vdash v : rel[l] * \Gamma' \qquad \vdash \Gamma \backslash \texttt{pc} \leq \Gamma'}{\Psi; \Gamma \vdash \texttt{jmpr}\ v}$$

9.1.3 Summary

This concludes the discussion of the abstract machine and type system we have chosen to use as our core TAL language. A formal description for the full language with templates and all the rules will be given with terse explanation at the end of the chapter. The rules without any explanation are summarized in Appendix B. Before continuing, we summarize some of our key design choices.

1. Machine states (M) consist of a heap, a register file, and a sequence of instructions to execute.

2. The machine semantics are given via a deterministic rewrite relation $M \longmapsto M'$. This relation manipulates typing annotations, but admits a type erasure interpretation as well. In other words, the typing annotations do not affect the outcome of computation.

3. Typed code sequences ($\texttt{code}\Gamma.I$) and word-sized values (w) are stored in the heap. The register file only stores word-sized values (w).

4. Word-sized values are all integers but various kinds of annotations are used to direct type checking. Word-sized values include integers (i), relative pointers ($\texttt{rel}[i] * l$), and singleton pointers ($\texttt{S}(l)$). Absolute pointers ($*l$) are just shorthand for $\texttt{rel}[0] * l$. For each of these values, there is a corresponding type.

5. Relative jumps and holes use offsets from a `pc` register to index into the heap. As a result, our abstract machine must model instruction sizes and support relative pointers. For the `pc` register to be useful, our type system must track its value precisely. We use singleton pointers for this purpose.

In previous work [MWCG99] it has been shown that this core TAL augmented with polymorphism, dynamic allocation, and tuples (but without relative jumps, and a `pc` register) is expressive enough to compile System F [Gir71, Rey74]. We omitted these features to simplify the extension to templates.

9.2 Overview of Templates

To support template-based RTCG, we add template values and types to TAL. We model template values as a contiguous subset of the heap that may contain code and data. Certain locations within the template are distinguished as template holes that must be filled with values of particular types before the body may be directly accessed.

Programmers generate code at run time by appending multiple copies of templates. To create control-flow between different copies, programmers fill holes with addresses. To embed values in the instruction stream, programmers fill holes with values. When all the holes have been filled, the contents of the template may be accessed as though they were part of the regular heap.

To support templates we add five new instructions to the abstract machine: 1) `start` allocates a new template, 2) `end` makes the body of a template available when all holes have been filled, 3) `copy` copies one template onto the end of another, 4) `fill` is used to fill holes with values, and 5) `fillr` is used to fill holes with relative addresses from within a template.

These instructions require significant changes to the type system.

1. The `start` instruction allocates a new template; therefore, the type system must support allocation.

2. The `copy` instruction appends a template onto another causing the type of one template to change. The TAL type system relies on values in the heap remaining type invariant.

3. The `copy` instruction, when used in a loop, may cause a template to grow without bound. To compile Cyclone we must support this feature. Because our abstract machine uses an integer address space, it may be necessary to move the template if it no longer fits in the locally available space. Therefore, templates must be *position independent*.

In the following subsections we describe how we address these issues before presenting the semantics of the instructions.

9.2.1 Type-Changing Operations

Operations that change types, such as template copying, interact poorly with aliasing. Suppose that in the current pre-condition Γ, $\Gamma(r_1) = *\sigma_1$ and $\Gamma(r_2) = *\sigma_2$. If σ_1 and σ_2 are template types then we might wish to `copy` the template in r_2 onto the end of the template in r_1. This operation would change the type of r_1 in a way inconsistent with its old type. If r_2 and r_1 are aliases then, to be consistent, the type of r_2 would have to change. However, the type system does not know the aliasing relationship and therefore cannot statically type check this operation.

Notice, that the types of registers do change freely. In this case type-changing operations can be checked because registers may not be aliased.

Our solution to this aliasing problem is to augment the domain of register files and register-file types with addresses in addition to registers. The type-checking rules will ensure that no pointers to addresses in the register file are created thus forbidding aliasing. To access the contents of these non-aliased addresses, programmers may only use singleton pointers. Singleton pointers make the aliasing relationship explicit thus keeping type-checking practical. In contrast, regular pointer types of the form $*\sigma$ do not make aliasing explicit so we refer to them as *anonymous pointers*.

When allocating a new template, the rewrite rules add its address to the register file. This address is by construction unaliased. Because the address is statically unknown, we introduce abstract type variables (ρ) to represent its location in the register-file type. The rule for the `start` instruction shows how this works.

$$\frac{\Gamma' = \{\Gamma, \rho \mapsto \sigma\}}{\{\Delta; \Psi; \Gamma\}\ \ \texttt{start}\ [\rho]r\ \{\Delta, \rho; \Psi; \Gamma'[r = S(\rho)]\}}$$

In this rule, σ is a placeholder for the canonical type of an empty template. (We will fill in this type later.) After executing this instruction, the programmer accesses the newly allocated template through the singleton pointer in register r.

Having added type variables to represent addresses (or equivalently type variables to represent integers), we must now add the other standard machinery of polymorphism.

1. Integer type variables (ρ or δ). As a convention, we use ρ for type variables representing addresses, and δ otherwise.

2. Type environments in judgments (Δ). All typing rules now have a set of bound type variables (Δ). For example, in the rule for `start` above we have already added Δ to the judgement for instructions.

3. Quantifiers for code types. Code types may now be polymorphic over type variables. The code value is augmented to be $\texttt{code}[\Delta]\Gamma.I$, and the code type becomes $\forall[\Delta].\Gamma$. In these types Δ is treated as an ordered list of type variables. In judgements we implicitly permit Δ to be reordered.

4. Type instantiations. Because code blocks are polymorphic, their type variables must be instantiated before jumping to them.

   The operand $v[\gamma]$ and the word value $w[\gamma]$ instantiate type variables. The typing rules for both are identical.

   $$\frac{\Delta; \Psi \vdash w : \mathit{rel}[\gamma']\ * \forall[\rho, \Delta'].\Gamma'}{\Delta; \Psi \vdash w[\gamma] : \mathit{rel}[\gamma']\ * \forall[\Delta'].\Gamma'(\rho \leftarrow \gamma)} (\rho \notin \Delta)$$

   (The notation $\Gamma'(\rho \leftarrow \gamma')$ denotes capture-avoiding substitution of γ' for ρ in Γ'.)

5. Integer type expressions (γ). We introduce a new syntactic class (γ) for integer type expressions – so far, either an integer (i) or an integer type

variable (ρ or δ). To support template copying we will later support addition ($\gamma_1 + \gamma_2$).

Where previously types included concrete integers, they may now include arbitrary integer type expressions (γ). For example, relative pointers used to be of the form $rel[i] * l$ but are now of the form $rel[\gamma] * l$. Similarly, singleton pointers have the form $S(\gamma)$.

It may appear that polymorphism has nullified the original intent of augmenting register-file types with addresses. The idea was to use singleton pointers to make aliasing relationships explicit. Consider a code block with the type $\forall[\rho, \rho'].\{\rho \mapsto \sigma, \rho' \mapsto \sigma\}$ where σ is a template type. In the body of this function we would like to copy the template pointed to by ρ' onto the end of the template in ρ. Again, it appears that ρ and ρ' might refer to the same location if ρ and ρ' are instantiated with the same location. If so, as in our earlier example, we cannot type check the template copy.

A first solution might attempt to forbid instantiations giving ρ and ρ' the same value. Such a solution requires a non-local check on every instantiation. We take a different approach with the same aim.

Rather than forbidding bad instantiations, the type system ensures that badly instantiated function pointers will not be useful. To do so, the type system treats register-file types as multi-sets of pairs. Useful types will always be partial functions in practice, but "bad" instantiations may create true multi-sets. For example, both ρ and ρ' could be instantiated with 4 to obtain the pre-condition:

$$\{4 \mapsto \sigma, 4 \mapsto \sigma\}$$

(assuming ρ and ρ' do not appear free in σ). This register-file type would be *neither* equal to, a subtype of, nor a supertype of $\{4 \mapsto \sigma\}$.

To ensure that code with this pre-condition is never used, the type system maintains the invariant that the register-file type in a well-typed machine state is always a partial function (it contains no duplicate elements in its domain). Because subtyping does not permit duplication, a control-transfer from a well-typed machine state to a function whose pre-condition is not a partial function is impossible. Therefore, a pointer with the "bad" pre-condition given above would either be unusable or ill-typed.

Now that we have added addresses to the register file there is no longer much distinction between register files and heaps. Therefore we use a single syntactic class for heaps and register files; and a single syntactic class for heap types and register-file types.

Figure 9.3 summarizes the changes to the type system and values described in this section. The additions to the type system we have described are a simplified version of our earlier work on alias types [SWM00, WM00]. Alias types rely on heap types, singleton pointers, and location polymorphism for the express purpose of permitting type-changing operations such as intializing newly allocated

(integers)	i, j, k, l		
(integer types)	γ	$::=$	$\delta \mid i$
(location types)	η	$::=$	$\gamma \mid r \mid \texttt{pc}$
(type envs)	Δ	$::=$	$\delta_1, \ldots, \delta_k$
(heap types)	Ψ, Γ	$::=$	$\{\eta_1 \mapsto \sigma_1, \ldots, \eta_n \mapsto \sigma_n\}$
(memory types)	σ	$::=$	$\tau \mid \forall[\Delta].\Gamma$
(word types)	τ	$::=$	$\mathit{int} \mid S(\gamma) \mid \mathit{rel}[\gamma] * \sigma \mid \mathit{junk}$
(locations)	L	$::=$	$i \mid r \mid \texttt{pc}$
(registers)	r	$::=$	$\texttt{r}_1 \mid \ldots \mid \texttt{r}_k$
(heaps)	H, R	$::=$	$\{L_1 \mapsto h_1, \ldots, L_n \mapsto h_n\}$
(heap values)	h	$::=$	$w \mid \texttt{code}[\Delta]\Gamma.I$
(word values)	w	$::=$	$i \mid \texttt{S}(i) \mid \texttt{rel}[\gamma] * i \mid w[\gamma]$
(operands)	v	$::=$	$r \mid w \mid [] \mid v[\gamma]$

Figure 9.3: Modifications of TAL/T required by polymorphism.

memory, and deallocating memory. The other features of alias types have been omitted (again) to simplify the presentation. The reader interested in more details on alias types may also wish to study the closely related logic of Bunched Implications [O'H93, O'H00, IO01].

9.2.2 Position Independence

Another important invariant of the type system is that templates are position independent. This property is important because the `copy` instruction may move a template if it cannot grow it in place.

Not all code is position independent. For example a relative jump `jmpr` 20, within a template, jumps to the same code after the template has been moved only if the offset 20 lies within the template. Similarly, an absolute jump `jmp` 20 will be unaffected by moving the template only if the absolute address 20 is outside of the template.

These examples demonstrate the two properties that guarantee position independence:

1. All jump instructions within a template to destinations in the same template must be relative.

2. All jumps from inside a template to destinations outside the template must be absolute.

We devise our template values and types to enforce these invariants by holding the exact location of the template abstract.

A template is written as $\texttt{tmpl}[\rho = l]\Psi.H$. Each piece of this term plays an important role:

- H is a contiguous piece of memory starting at location 0. The function $\mathrm{dom}^+ H$ (formally defined later) returns the set of locations occupied by H. Using this function we can express this contiguity condition more formally as $\mathrm{dom}^+ H = \{i | 0 \leq i \leq k\}$.

- l represents the template's base address in memory. In other words, the template takes up locations l through $l + k$ (where k is from the example above). The operation $H + j$ increments every element in the domain of H by j. We can similarly define $\Psi + \gamma$ to increment every element in the domain of Ψ by γ. For this operation to make sense we extend integer type expressions (γ) to include the additional form $\gamma_1 + \gamma_2$.

  Once all the holes have been filled, the `end` instruction can be used to convert this template into the heap $H + l$.

- Ψ indicates the types and locations of holes within H. As holes are filled, using the `fill` and `fillr` instructions, Ψ will shrink.

- ρ represents l during type-checking. Changing l will not affect the type-checking of the template in any way; hence the template will be position independent. The occurence of ρ in the template value binds ρ in Ψ and H.

A template type is simply $[\rho]\Psi \Rightarrow \Psi'$ where Ψ is the type of the holes, and Ψ' is the type H will have once all the holes have been filled.

The following example shows a template containing a hole at offset 2 that must be filled with a code pointer. The code in this template doubles its argument.

$$\begin{array}{l}\texttt{tmpl}[\rho = 10]\{2 \mapsto *\forall[].\{\texttt{r}_1 \mapsto \mathit{int}\}\}. \\ \quad \{0 \mapsto \texttt{code}[]\{\texttt{r}_1 \mapsto \mathit{int}, \texttt{pc} \mapsto S(\rho)\}.\ \texttt{add}\ \texttt{r}_1, \texttt{r}_1, \texttt{r}_1; \texttt{jmp}\ [], \\ \quad 2 \mapsto 0\}\end{array}$$

The hole is currently unfilled so location 2 holds an arbitrary value. This template would have type

$$[\rho]\{2 \mapsto *\forall[].\{\texttt{r}_1 \mapsto \mathit{int}\}\} \Rightarrow \{0 \mapsto \forall[].\{\texttt{r}_1 \mapsto \mathit{int}, \texttt{pc} \mapsto S(\rho)\}$$

Filling the hole with a valid pointer would change the template's type to

$$[\rho]\emptyset \Rightarrow \{0 \mapsto \forall[].\{\texttt{r}_1 \mapsto \mathit{int}, \texttt{pc} \mapsto S(\rho)\}$$

And finally, the `end` instruction could be used to extend the current heap type with the heap type

$$\{\rho + 0 \mapsto \forall[].\{\texttt{r}_1 \mapsto \mathit{int}, \texttt{pc} \mapsto S(\rho)\}$$

The `end` instruction will also leave a singleton pointer $S(\rho)$ in a register so that this code may be accessed.

Notice that the `pc` in the template contains a singleton pointer to location ρ. When the template is opened by the `end` instruction, this value is the only value of `pc` that will allow the resulting code to be used.

This invariant is subtle and important. The reason we introduced the function $\sigma@l$ was to eliminate `pc` from pre-conditions but only when the `pc` matched the location of the code. Note also that the rule for type-checking `jmp` requires an anonymous pointer $(*\Gamma)$ whose pre-condition does not contain `pc` in its domain. The function $\sigma@l$ will only allow `pc` to be removed (in this case) if it is $S(\rho)$.

In fact, all useful code within the template will have a `pc` with type $S(\rho+i)$ for some i. Because ρ is abstract, programs cannot manufacture relative pointers to addresses outside the template. Similarly absolute jumps can only be made outside the template because internal locations are all relative to ρ.

The rule for type-checking template values ensures that templates are position independent.

$$\text{(h-tmpl)}\frac{\begin{array}{c}\Delta,\rho;\{\Psi,\Psi_1+\rho,\Psi_2+\rho\}\vdash H:\{\Psi_1',\Psi_2\} \qquad \Delta\vdash\{\Psi_1,\Psi_2\}\\ \mathrm{dom}^+H=\{i|0\le i\le k\} \qquad \mathrm{dom}\ \Psi_1=\mathrm{dom}\Psi_1'\end{array}}{\Delta;\Psi\vdash \mathtt{tmpl}[\rho=l]\Psi_1.H:[\rho](\Psi_1\Rightarrow\Psi_2)}(\rho\notin\Delta)$$

In this rule, H is type-checked as though all the holes have been filled and the `end` instrution already issued. This fact is reflected in the assumption $\{\Psi,\Psi_1+\rho,\Psi_2+\rho\}$ where Ψ_1 contains the types of the holes and Ψ_2 contains the recursive type of the template body. The type variable ρ represents the abstract offset at which the template will be opened. No requirement is placed on the resulting type of the unfilled holes in H. We only insist that the locations of the holes are part of the domain of H through the requirement $\mathrm{dom}\Psi_1=\mathrm{dom}\Psi_1'$.

9.2.3 Template Instructions

With all the necessary typing mechanism in place we can now present the operational semantics for the template-specific instructions. To help keep these operational rules small we elide updating the `pc` register by the size of the instruction in each case.

To allocate a new instruction, the programmer uses the `start` instruction.

$$(H,R,\mathtt{start}\ [\rho]r;I)\ \longmapsto\ (H,\{R[r=\mathtt{S}(l)],l\mapsto\mathtt{tmpl}[\rho'=i']\emptyset.\emptyset\},I(\rho\mapsto l))$$

where i' and l are newly allocated locations. The type variable ρ represents the newly allocated location. The operational instructions substitute l for ρ throughout the remaining instructions in the block reflecting this fact. The location i' is a new location that serves as the base address for the template.

The above rule does not yet specify how to pick l or i'. Rather than choose a particular allocation strategy we use a non-determinstic allocator. The function $malloc_{H,R}(k)$ allocates k consecutive addresses not in $\mathrm{dom}^+ H$ or $\mathrm{dom}^+ R$. To simplify the presentation we let $l = malloc_{H,R}(2)$ and set $i' = l + 1$.

The `end` instruction is used, when all holes have been filled, to make the body of the template available for use.

$$\begin{gathered}(H, \{R, r \mapsto \mathtt{S}(l), l \mapsto \mathtt{tmpl}[\rho' = i]\emptyset.H_2\}, \mathtt{end}\ [\rho]r; I) \longmapsto \\ (\{H, H_2 + i(\rho' \leftarrow i)\}, \{R, r \mapsto \mathtt{S}(i)\}, I(\rho \leftarrow i))\end{gathered}$$

The only surprising aspect of this rule should be the indirection through $\mathtt{S}(l)$. It is necessary because values in registers can be copied (via the `mov` instruction), but heap values in addresses cannot. Because copying is impossible, there are no aliases to locations in R making it possible to consistently and safely change the type of the template when holes are filled, or new templates are appended to it.

The `copy` $[\delta]r_1, r_2, v$ instruction is used to copy one template onto the end of another. Intuitively, this instruction appends a copy of the template in v into the template "in" r_1. It leaves the offset of the copied template as a singleton integer (with type $S(\delta)$) in register r_2. This offset will be useful for filling holes and creating control-flow between the two copies.

$$(H, R, \mathtt{copy}\ [\delta]r_1, r_2, v) \longmapsto (H, R[l_1 = h_1][r_2 = \mathtt{S}(k)], I(\delta \leftarrow k))$$

when all the following hold,

1. $R(r_1) = \mathtt{S}(l_1)$ and $R(l_1) = \mathtt{tmpl}[\rho_1 = i_1]\Psi_1.H_1$

2. $S_{H,R} = *l_3$ and $H(l_3) = \mathtt{tmpl}[\rho_3 = i_3]\Psi_3.H_3$

3. $h_1 = (\mathtt{tmpl}[\rho_1 = i]\{\Psi_1, \Psi_3 + k\}.\{H_1, H_3 + k\})(\rho_3 \leftarrow \rho_1 + k)$ where k is the largest address in $\mathrm{dom}^+ H_1$, and i is i_1 if the template can be grown in place or a new location if it needs to be moved.

   The formal rule uses another non-deterministic allocator *realloc* that tries to grow the template in place if possible.

The two remaining fill instructions provide support for embedding values and absolute jumps (`fill`), and support for creating control-flow between copies of templates (`fillr`).

The `fill` r_1, r_2, r_3 instruction is one of the simplest. It simply embeds the value in r_3 at offset r_2 within the template "in" r_1.

$$(H, R, \mathtt{fill}\ r_1, r_2, r_3; I) \longmapsto (H, R[l = \mathtt{tmpl}[\rho' = i'](\Psi' \backslash i).H'], I)$$

provided that $R(r_1) = \mathtt{S}(l)$, $R(r_2) = \mathtt{S}(i)$, $R(r_3) = w$, and $R(l) = \mathtt{tmpl}[\rho' = i']\Psi'.H'$. Notice that the hole is removed from the type of the template too.

The relative fill instruction (`fillr` r_1, r_2, r_3) is used for creating control-flow between copies within a template. The value in r_2 is the offset of the hole. The value in r_3 is the offset of some code. We will fill the hole with a relative pointer to this code.

$$\begin{array}{l}(H, R, \texttt{fillr}\ r_1, r_2, r_3; I) \longmapsto \\ \qquad (H, R[l = \texttt{tmpl}[\rho' = i'].(\Psi' \backslash i_2).H'[i_2 = \texttt{rel}[\rho + i_2] * i_3 - i_2]], I)\end{array}$$

where $R(r_1) = \texttt{S}(l_1)$, $R(r_2) = \texttt{S}(i_2)$, and $R(r_2) = \texttt{S}(i_3)$. Furthermore $R(l_1) = \texttt{tmpl}[\rho' = i']\Psi'.H'$.

Without the relative fill instruction it would not be possible to create cycles between copies of templates in generated code. All the typing mechanisms developed in this section take part in ensuring that this operation can be typed: singleton integers, relative pointers, and template types.

The type-checking rules for the RTCG instructions statically simulate the operational semantics directly. The key property of the type system we have described in this section is that such a static simulation remains consistent under evaluation.

Two more instructions are needed to permit writing useful programs in our formalism. Without additional support, holes cannot be filled after a `copy`, and anonymous pointers to generated code cannot be created.

Without a means of constructing interesting singleton integers the above fill instructions are useless. For example a template may contain a hole at offset 3. After copying this template into another, the `copy` instruction leaves the offset of the copy in a register – say with type $S(\delta)$. To fill the hole after the copy, we need a singleton pointer of type $S(\delta + 3)$ but have no way of creating one. The obvious solution is to support addition of singletons via a second operational rule for addition.

$$(H, R, \texttt{add}\ r_1, r_2, v) \longmapsto (H, R[r_1 = \texttt{S}(i_2 + i_3)], I)$$

where $R(r_2) = \texttt{S}(i_2)$ and $S_{H,R}(v) = \texttt{S}(i_3)$. With this addition, it is possible to fill any hole after a copy.

The results of `end` are also not useful without additional support. Consider the type-checking rule for `end`.

$$\frac{\Gamma(r) = S(\gamma) \qquad \Gamma = \{\Gamma', \gamma \mapsto [\rho'].(\emptyset \Rightarrow \Psi')\}}{\{\Delta; \Psi; \Gamma\}\ \ \texttt{end}\ [\rho]r\ \{\Delta, \rho; \{\Psi, \Psi' + \rho(\rho' \leftarrow \rho)\}; \Gamma'[r = S(\rho)]\}}(\rho \notin \Delta)$$

The heap type is extended with locations of the form $\rho + i$. It is impossible for the programmer to construct anonymous pointers $(*\sigma)$ into this region because ρ is abstract. But the jump instructions alls require anonymous pointers to their destination. Therefore we introduce the instruction `forget` to convert singleton pointers $(\texttt{S}(\gamma))$ into anonymous pointers. The operational rule is simple.

$$(H, R, \texttt{forget}\ r, v; I) \longmapsto (H, R[r = *l], I)$$

where $S_{H,R}(v) = \texttt{S}(l)$. The type-checking rule for `forget` statically simulates its execution.

$$\frac{\begin{array}{c}\Delta; \Psi; \Gamma \vdash v : S(\gamma) \\ \Psi(\gamma) = \sigma \qquad \sigma @ \gamma = \sigma'\end{array}}{\{\Delta; \Psi; \Gamma\}\ \ \texttt{forget}\ r, v\ \{\Delta; \Psi; \Gamma[r = *\sigma']\}}$$

Using this instruction the programmer can create anonymous pointers into the extended portions of the heap introduced by ending a template.

The instruction `forget` and addition for singleton pointers are standard instructions. They have nothing in particular to do with RTCG. We include them so that our formalism is useful and our soundness proof meaningful.

9.2.4 Summary

This completes the description of the type system and all the RTCG support.

- We have extended register-file types with addresses in addition to registers so that type-changing operations may be checked. This change also necessitated adding singletons.
- Handling allocation required the introduction of integer polymorphism. This choice created problems with register file types so we extended those to be multi-sets (instead of just partial functions).
- We introduced values for templates ($\texttt{tmpl}[\rho = i]\Psi.H$) and template types ($[\rho]\Psi_1 \Rightarrow \Psi_2$). In the template value the integer (i), serves as the base address for the heap in the body, and he heap type (Ψ) indicates the locations and types of holes.
- There are five RTCG specific instruction – `start`, `end`, `copy`, `fill`, and `fillr`. To support hole filling we also extended the `add` instruction to permit adding singleton integers. We introduce the `forget` instruction to convert singleton pointers into anonymous pointers.

9.3 TAL/T

This section presents the complete TAL/T language, operational semantics, and type system. It relies on the groundwork laid in previous sections to provide intuition and is therefore quite terse. The purpose of this section is to formalize the intuitive concepts explained earlier.

9.3.1 Syntax

Without RTCG. Figure 9.4 shows the syntax for the core part of the formalism without any of the RTCG additions. A machine state M consists of three components: 1) the heap (H), 2) the register file (R), and 3) the instruction sequence being executed (I).

Both the heap and the register-file are the same kind of object. They are both partial functions from locations (L) to heap values (h). Like in a real machine, locations are either an integer i or a register r. The type system will only permit word-sized values to be stored in registers.

(integers)	i, j, k, l		
(integer types)	γ	$::=$	$\delta \mid i \mid \gamma_1 + \gamma_2$
(location types)	η	$::=$	$\gamma \mid r \mid \texttt{pc}$
(type envs)	Δ	$::=$	$\delta_1, \ldots, \delta_k$
(heap types)	Ψ, Γ	$::=$	$\{\eta_1 \mapsto \sigma_1, \ldots, \eta_n \mapsto \sigma_n\}$
(memory types)	σ	$::=$	$\tau \mid \forall[\Delta].\Gamma$
(word types)	τ	$::=$	$\mathit{int} \mid S(\gamma) \mid \mathit{rel}[\gamma] * \sigma \mid \mathit{junk}$
(locations)	L	$::=$	$i \mid r \mid \texttt{pc}$
(registers)	r	$::=$	$\texttt{r}_1 \mid \ldots \mid \texttt{r}_k$
(heaps)	H, R	$::=$	$\{L_1 \mapsto h_1, \ldots, L_n \mapsto h_n\}$
(heap values)	h	$::=$	$w \mid \texttt{code}[\Delta]\Gamma.I$
(word values)	w	$::=$	$i \mid \texttt{S}(i) \mid \texttt{rel}[\gamma] * i \mid w[\gamma]$
(operands)	v	$::=$	$r \mid w \mid [] \mid v[\gamma]$
(instructions)	ι	$::=$	$\texttt{add}\ r_1, r_2, v \mid \texttt{bnz}\ r, v \mid \texttt{mov}\ r, v \mid$ $\texttt{forget}\ r_1, r_2, v$
(blocks)	I	$::=$	$\iota; I \mid \texttt{jmp}\ v \mid \texttt{jmpr}\ v$
(machine states)	M	$::=$	(H, R, I)

Figure 9.4: TAL/T syntax without RTCG

The type system maintains the invariant that types of locations in the heap (H) cannot change, whereas those in the register-file (R) may. By analogy with linear type systems, one can think of the heap as the non-linear part of the state, and the register-file as the linear part.

Registers range over a set of fixed built-in registers $\texttt{r}_1$ through $\texttt{r}_k$. There is also a distinguished program-counter register pc. The pc contains a pointer to the next instruction to be executed.

Unlike a real machine, heap values are not 32-bit integers. Each heap value h is either a word-size value (w), or a code block (denoted $\texttt{code}[\Delta]\Gamma.I$). A code block is a sequence of assembly instructions terminated by a control transfer instruction. The scope of the bound type variables (Δ) extends throughout the block (I) and the block's pre-condition (Γ).

Word values w are all integers but they come in different forms: plain integers (i), singleton pointers ($\texttt{S}(i)$), relative pointers ($\texttt{rel}[\gamma] * i$), and type instantiations ($w[\chi]$). Singleton pointers (or equivalently singleton integers) behave just like regular integers except that the type system tracks their values. An integer i will have type *int*, but a singleton integer $\texttt{S}(i)$ will have type $S(i)$.

A relative pointer ($\texttt{rel}[\gamma]*i$) is a generalization of anonymous pointers ($*i$). An anonymous pointer $*i$ is a pointer to some heap value at location i. The relative pointer $\texttt{rel}[\gamma] * i$ is an offset such that $i + \gamma$ points to a heap value. The final word values are instantiations of the form $w[\chi]$. These are used to instantiate the type variables bound to code pointers.

$$
\begin{array}{llcl}
\text{(memory types)} & \sigma & ::= & \ldots \mid [\rho].\Psi_1{\Rightarrow}\Psi_2 \\
\\
\text{(heap values)} & h & ::= & \ldots \mid \texttt{tmpl}[\rho = i]\Psi.H \\
\text{(instructions)} & \iota & ::= & \ldots \mid \texttt{start}\ [\rho]r \mid \texttt{end}\ [\rho]r \mid \texttt{copy}\ [\delta]r_1, r_2, v \mid \\
 & & & \texttt{fill}\ r_1, r_2, v \mid \texttt{fillr}\ r_1, r_2, r_3 \mid
\end{array}
$$

Figure 9.5: TAL/T syntax additions for RTCG

The abstract machine supports a small set of instructions. Each instruction operates only on registers or operands. An operand v is either a word value w, a register r, a hole [], or an instantiated operand $v[\chi]$. A hole allows an instruction to read a value relative to the `pc` register. We delay discussing the meanings of the instructions until we present the operational semantics.

So far we have explained all the syntactic categories except the various kinds of types. Types are split into three levels: 1) heap types (Ψ or Γ), 2) memory types (σ), and 3) word types (τ). Memory types are used for values in the heap. Word types are used to type word-sized values that can be stored in registers.

Well-formed heap types Ψ are partial functions from locations (either registers or addresses represented by integer type expressions) to heap types (σ). For technical reasons, heap types in general are mult-sets; but, for practical purposes they are always partial functions. By convention we use Γ to denote a heap corresponding to a register file, and Ψ for heaps.

The only interesting memory type is the type for code $\forall[\Delta].\Gamma$. This type indicates a polymorphic function quantified over the ordered list of variables Δ. Type variables (δ or ρ) range over integers. Γ denotes the type the linear part of the heap must assume before this code may be executed. In other words, Γ is the pre-condition for the register file. The pre-condition for the heap is implicit because the types of heap elements are invariant.

Word types include many different forms. The singleton pointer type ($S(\gamma)$) designates the unique address represented by the integer type expression γ. The relative pointer type $rel[\gamma] * \sigma$ is a pointer to an anonymous location i such that $\gamma + i$ has type σ. This type will be most used when filling relative jump holes in templates.

We introduce a type *junk* that can be assigned to any word-sized value. This type is usefully assigned to registers in a pre-condition when their type does not matter. The type *junk* is a super-type of every other word type.

An integer type expression γ is either the sum of two integer type expressions ($\gamma_1 + \gamma_2$), an integer type variable (δ or ρ), or an integer (i).

A singleton pointer type $S(\gamma)$ indicates that the value with this type is a pointer to location γ. When combined with polymorphism, singleton pointers, integer type expressions, and heap types provide a flexible means for expressing pre-conditions.

Additions for RTCG. To support template-based RTCG, we augment the core language with a primitive template construct and instructions for manipulating templates. Figure 9.5 shows all the additions.

The most important addition is the new heap value for templates. A template is a contiguous subset of the heap that can be copied, and contains holes to be filled. This fact is reflected in the syntactic form for templates $\texttt{tmpl}[\rho = i]\Psi.H$. The type variable ρ abstracts the current location of the template indicating that the heap H can be moved while remaining well typed. The integer i is the base address of the template. And, Ψ indicates the types and locations of the holes to be filled in H.

TAL/T programs use templates to generate new code. They can allocate new templates with the `start` instruction, concatenate templates using the `copy` instruction, and fill holes in templates with the `fill` and `fillr` instructions. Finally, when all the holes have been filled, the generated code can be made accessible using the `end` instruction.

Many of our design choices were forced by a desire to model templates accurately. The choice to invoke the machinery of alias types was motivated by the need to handle the changing type of a template as template copies are appended to it and holes are filled. Without some form of linearity, static type systems cannot handle values of changing types.

We also introduced relative jumps, and integer locations to better model templates. For example, the implementation moves templates when there is not enough space to perform a `copy`. Therefore it is important that templates be position independent. Position independence is guaranteed by two properties: 1) jumps to labels outside a template must be absolute, and 2) jumps within a template must be relative. In previous versions of TAL, locations were abstract labels and there were no relative jumps so position independence could not be expressed within the language. We introduced relative pointers to type-check the operand in a relative jump instruction.

When copying templates together, TAL/T programs should be able to create intra-template control flow. After copying two templates together it should be possible to create relative jumps between the two copies. This requiremenet motivated the introduction of the integer type expression $\gamma_1 + \gamma_2$. Intuitively each copy is placed at some offset in the enclosing template. The relative fill instruction `fillr` makes it possible to fill a hole with a pointer relative to the location of the hole, and the location of the label.

It is also important for TAL/T programs to be able to forget parts of templates they have copied. Otherwise, a TAL/T program could only copy a statically known number of templates together. Therefore, our type system allows subtyping on templates so that generated code can be arbitrarily large.

9.3.2 Notation

In the definition of the operational semantics and the typing judgements we rely on the following notation and implicit conventions. The notation $X(a \leftarrow v)$ denotes the capture-avoiding substitution in X of all occurences of a by v.

Our proof relies on a strong notion of equality for syntactic objects. As in most proofs of this kind, we consider α-equivalent terms to be equal. That is the two syntactic expressions $\forall\alpha.\alpha$ and $\forall\beta.\beta$ denote the same underlying semantic object. For our purposes, we further enrich the notion of implicit equality. First integer type expressions are implicitly reordered and simplified. Therefore the two expression $4 + 3$ and 7 will denote the same object, as will $\delta_1 + \delta_2$ and $\delta_2 + \delta_1$. Second, syntactic forms of the shape $\{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\}$ denote multi-sets of pairs. These include the syntactic classes Ψ, Γ, H, R. For these terms order does not matter.

We adopt the following notation to manipulate multi-sets (denoted by S below):

$\emptyset$	The empty multi-set.
$\{S_1, S_2\}$	Union of multi-sets S_1 and S_2.
$S[x = y]$	Pick one occurence of $x \mapsto z$ in S, and replace it with $x \mapsto y$ Undefined if there is no occurence.
$S(x)$	Any element y such that $S = \{S', x \mapsto y\}$. Undefined if no such y exists.
$\mathrm{dom}S$	The domain of S (the set $\{x \vert x \mapsto z \in S\}$)
$S \backslash x$	The multi-set $\{y \mapsto z \vert y \mapsto z \in S, y \neq x\}$.

In all cases, well-formed types will only contain partial functions. However substitution may turn a partial function into a multi-set. For example, substituting 3 for ρ in $\{3 \mapsto \mathit{int}, \rho \mapsto \mathit{int}\}$ will turn this partial function into a multi-set. Such substitutions will be legal but the resulting value will never be useful. We adopt non-deterministic semantics for $S(x)$ and $S[x = y]$ to simplify the proof when true multi-sets are created by substitution.

9.3.3 Operational Semantics

The operational semantics define a deterministic rewrite relation on machine states (H, R, I). The operational rules we present maintain type information to simplify the proof, but admit a type-erasure interpretation. (The types can be erased without affecting the outcome.)

The rules also place requirements on the shapes of certain objects. For example, the addition rule requires that $R(r_2) = i$. If this condition fails to hold the rule will not apply, and the machine will be stuck. In each case, operational rules only apply if all conditions are met.

For instructions that access operands v, the function $S_{H,R}(v)$ is used to compute

$\boxed{\mathrm{dom}^+ H,\ |H|^+}$

$$\begin{array}{rcl} \mathrm{dom}^+\{H_1, H_2\} & = & \mathrm{dom}^+ H_1 \cup \mathrm{dom}^+ H_2 \\ \mathrm{dom}^+\{l \mapsto \mathtt{code}[\Delta]\Gamma.I\} & = & W(l, I) \\ \mathrm{dom}^+\{l' \mapsto \mathtt{tmpl}[\rho = l]\Psi_1.H\} & = & \{l'\} \cup \{l + i | i \in \mathrm{dom}^+ H\} \end{array}$$

where

$$\begin{array}{rcl} W(l, \iota; I) & = & \{l\} \cup W(l + |\iota|, I) \\ W(l, \mathtt{jmp}\ r) & = & \{l\} \end{array}$$

$|H|^+ = k$ whenever $\mathrm{dom}^+ H = \{i | 0 \leq i < k\}$.

Figure 9.6: Definition of extended domains for heaps.

$$\frac{\{l + i | 0 \leq i < k\} \notin \mathrm{dom}^+\{H, R\}}{malloc_{H,R}(k) = l}$$

$$\frac{\{i_b + j | 0 \leq j < k_f\} \notin (\mathrm{dom}^+\{H, R\} \backslash \{i_b + j | 0 \leq j < k_i\})}{realloc_{H,R}(i_b, k_i, k_f) = l}$$

Figure 9.7: Non-deterministic memory allocators

the word represented by the operand.

$$S_{H,R}(v) = \begin{cases} R(r) & \text{when } v = r \\ w & \text{when } v = w \\ S_{H,R}(v')[\chi] & \text{when } v = v'[\chi] \\ w & \text{when } v = [] \text{ and } R(\mathtt{pc}) = \mathtt{S}(i) \\ & \quad \text{and } H(i+1) = w \end{cases}$$

Because our abstract machine models holes in instruction streams, and uses integers for addresses; it must include a notion of size. The non-jumping instructions all increment the `pc` register by their size. We chose a simple model for instruction sizes. Instructions that do not contain holes have size 1; those that do have size 2. Formally, we use the function $|\iota|$ to compute the size of an instruction.

$$|\iota| = \begin{cases} 1 + |v| & \text{when } v \text{ occurs in } \iota \\ 1 & \text{otherwise} \end{cases} \quad \text{and } |v| = \begin{cases} 1 & \text{when } v = [] \\ |v'| & \text{when } v = v'[\chi] \\ 0 & \text{otherwise.} \end{cases}$$

Similarly word sized values have size one. The domain of a heap H is the set of integer locations containing heap values. The extended domain of a heap H, written $\mathrm{dom}^+ H$, is the set of allocated memory locations. And $|H|^+$ is the maximum allocated address. These two functions are formally defined in Figure 9.6. Notice that code blocks occupy a range of memory addresses.

To allocate memory, we define two allocators shown in Figure 9.7. These functions non-deterministically return a range of free memory addresses. The function

$\boxed{M \longmapsto M'}$ (no RTCG)

For the non-rtcg instructions where $(H, R, \iota; I) \longmapsto (H', R'[\mathtt{pc} = \mathtt{S}(i + |\iota|)], I')$ where $R(\mathtt{pc}) = \mathtt{S}(i)$:

$\iota =$	$H' =$	$R' =$	$I' =$
$\mathtt{add}\ r_1, r_2, v$	H	$R[r_1 = R(r_2) + S_{H,R}(v)]$	I
$\mathtt{add}\ r_1, r_2, v$	H	$R[r_1 = \mathtt{S}(i + j)]$	I
when $R(r_2) = \mathtt{S}(i)$ and $S_{H,R}(v) = \mathtt{S}(j)$			
$\mathtt{bnz}\ r, v$	H	R	I
when $R(r) = 0$			
$\mathtt{mov}\ r, v$	H	$R[r = S_{H,R}(v)]$	I
$\mathtt{forget}\ r, v$	H	$R[r = *i]$	I
when $S_{H,R}(v) = \mathtt{S}(i)$			

Figure 9.8: Operational rules for the core TAL/T

$malloc_{H,R}(k)$ returns some address l such that l through $l + k - 1$ are currently unallocated. The function $realloc_{H,R}(i, k_1, k_2)$ is used by template copy to grow a template in place. The three arguments represent the starting address of the currently allocated memory, the size of the allocated block, and the desired size. If the object cannot be grown in place a new memory address is returned.

Without RTCG. Figure 9.8 shows the operational rules for the non-jumping instructions, and Figure 9.9 contains the operational rules for the control-transfer instructions.

Below we describe their intended semantics.

1. `add` r_1, r_2, v (1) adds the operand v to the contents of register r_2 and writes the result into register r_1. This is the only arithmetic operation supported by our machine. Generalizing to other arithmetic operations is straightforward.

2. `add` r_1, r_2, v (2) is similar to `add` (1) but allows singletons to be added.

3. `bnz` r, v branches to the location pointed to by v if r is not zero.

4. `mov` r, v move the value represented by v into register r

5. `forget` r, v converts the singleton pointer that v represents into an anonymous pointer in register r.

6. `jmp` v jumps to the location pointed to by $S_{H,R}(v)$.

$\boxed{M \longmapsto M'}$ (control instructions)

For $(H, R, I) \longmapsto M'$.

$I =$	$M' =$
$\texttt{bnz}\ r, v; I'$	$(H, R[\texttt{pc} = \texttt{S}(l)], I''(\delta_1, \ldots, \delta_n \leftarrow \gamma_1, \ldots, \gamma_n))$
	where $S_{H,R}(v) = *l[\gamma_1] \ldots [\gamma_n]$
	and $H(l) = \texttt{code}[\delta_1, \ldots, \delta_n]\Gamma.I''$ and $R(r) \neq 0$
$\texttt{jmp}\ v$	$(H, R[\texttt{pc} = \texttt{S}(l)], I'(\delta_1, \ldots, \delta_n \leftarrow \gamma_1, \ldots, \gamma_n))$
	where $S_{H,R}(v) = *l[\gamma_1] \ldots [\gamma_n]$ and $H(l) = \texttt{code}[\delta_1, \ldots, \delta_n]\Gamma''.I'$
$\texttt{jmpr}\ v$	$(H, R[\texttt{pc} = \texttt{S}(l)], I'(\delta_1, \ldots, \delta_n \leftarrow \gamma_1, \ldots, \gamma_n))$
	where $S_{H,R}(v) = (\texttt{rel}[\gamma] * i)[\gamma_1] \ldots [\gamma_n]$ and $R(\texttt{pc}) = \texttt{S}(l')$
	and $l = l' + i$ and $H(l) = \texttt{code}[\delta_1, \ldots, \delta_n]\Gamma.I'$

Figure 9.9: Operational rules for the TAL/T control transfer instructions.

7. `jmpr` v jumps relative to the `pc` register. If $S_{H,R}(v) = \texttt{S}(i)$ and $R(\texttt{pc}) = \texttt{S}(j)$ then this instruction jumps to the address $i + j$.

Additions for RTCG. Each of the five RTCG instructions play an important role in the life of a template. Figure 9.10 shows their formal semantics.

1. `start` $[\rho]r$ allocates a new template and places a singleton pointer to it in register r. The type variable ρ is bound to this new location in the subsequent code.

2. `end` $[\rho]r$ assumes r contains a singleton pointer to a template with no holes. This template is therefore complete and its contents can now be made available for use. After this instruction executes r will contain a singleton pointer to the base address of the now opened template. The type variable ρ is bound to this base address.

3. `copy` $[\delta]r_1, r_2, v$ copies the template pointed to by v after the template pointed to by r_1. After this instruction executes, the register r_2 contains a singleton integer with type $S(\delta)$ that is the offset of the copy of the template from v in the template pointed to by r_1. The `copy` instruction is used to assemble new code.

4. `fill` r_1, r_2, v fills the hole in the template pointed to by r_1 at offset r_2 with the value v.

5. `fillr` r_1, r_2, v is similar to the `fill` instruction but is used to fill a hole with an offset from within r_2. This is used primarily to create intra-template jumps.

$\boxed{M \longmapsto M'}$ (RTCG)

For the rtcg instructions where $(H, R, \iota; I) \longmapsto (H', R'[\mathtt{pc} = \mathtt{S}(i + |\iota|)], I')$ where $R(\mathtt{pc}) = \mathtt{S}(i)$:

$\iota =$	$H' =$	$R' =$	$I' =$
$\mathtt{start}\ [\rho]r$	H	$\{R[r = \mathtt{S}(i)], i \mapsto h\}$	$I(\rho \leftarrow i)$
when $i = malloc_{H,R}(2)$, and $i' = i + 1$			
and $h = \mathtt{tmpl}[\rho' = i']\emptyset.\emptyset$			
$\mathtt{end}\ [\rho]r$	$\{H, H_2 + i(\rho' \leftarrow i)\}$	$R''[r = \mathtt{S}(i)]$	$I(\rho \leftarrow i)$
when $R(r) = \mathtt{S}(l)$ and $R = \{R'', l \mapsto \mathtt{tmpl}[\rho' = i]\emptyset.H_2\}$			
$\mathtt{copy}\ [\delta]r_1, r_2, v$	H	$R[l_1 = h_1][r_2 = \mathtt{S}(k)]$	$I(\delta \leftarrow k)$
when $R(r_1) = \mathtt{S}(l_1)$ and $R(l_1) = \mathtt{tmpl}[\rho_1 = i_1]\Psi_1.H_1$			
and $S_{H,R}(v) = *l_3$ and $H(l_3) = \mathtt{tmpl}[\rho_3 = i_3]\Psi'_1.H_3$			
and $H' = \{H_1, H_3 + k\}$and $k = \lvert H_1\rvert^+$ and $i = realloc_{H,R}(i_1, k, \lvert H'\rvert^+)$			
and $h_1 = \mathtt{tmpl}([\rho_1 = i]\{\Psi_1, \Psi'_1 + k\}.H')(\rho_3 \leftarrow \rho_1 + k)$			
$\mathtt{fill}\ r_1, r_2, r_3$	H	$R[l = h]$	I
when $R(r_1) = \mathtt{S}(l)$ and $R(r_2) = \mathtt{S}(i)$ and $R(r_3) = w$			
and $R(l) = \mathtt{tmpl}[\rho = i']\{\Psi_1, i \mapsto \tau\}.H'$			
and $h = \mathtt{tmpl}[\rho = i']\Psi_1.H'[i = w]$			
$\mathtt{fillr}\ r_1, r_2, r_3$	H	$R[l_1 = h]$	I
when $R(r_1) = \mathtt{S}(l_1)$ and $R(r_2) = \mathtt{S}(i_2)$ and $R(r_3) = \mathtt{S}(i_3)$			
and $R(l_1) = \mathtt{tmpl}[\rho = i']\{\Psi_1, i_2 \mapsto \tau\}.H'$			
and $h = \mathtt{tmpl}[\rho = i']\Psi_1.H'[i_2 = \mathtt{rel}[\rho + i_2] * (i_3 - i_2)]$			

Figure 9.10: Operational rules for the TAL/T RTCG instructions.

$\vdash M$	M is a well-formed and well-typed program state.
$\Delta; \Psi \vdash H : \Psi'$	H is well-formed and has type Ψ'.
$\Delta; \Psi \vdash h : \sigma$	h is well-formed and has type σ.
$\Delta; \Psi \vdash w : \tau$	w is well-formed and has type τ.
$\Delta; \Psi; \Gamma \vdash v : \tau$	v is well-formed and has type τ.
$\Delta; \Psi; \Gamma \vdash I$	The block I is well-formed and well-typed.
$\{\Delta; \Psi; \Gamma\}\ \iota\ \{\Delta'; \Psi'; \Gamma'\}$	When ι is executed in a state with type Δ, Ψ, Γ it produces a state with type Δ', Ψ', Γ'.
$\Delta \vdash \Psi_1 \leq \Psi_2$	Ψ_1 is a sub-heap of Ψ_2.
$\Delta \vdash \sigma_1 \leq \sigma_2$	σ_1 is a sub-type of σ_2.
$\sigma@\gamma = \sigma'$	When at location γ, σ is equal to σ'.
$\Delta \vdash \Psi$	The heap type Ψ is well-formed.
$\Delta \vdash \sigma$	The type σ is well-formed.

Figure 9.11: Collection of TAL/T judgement forms.

9.3.4 Static Semantics

TAL/T contains many syntactic categories and therefore requires many kinds of judgements. These are summarized in Figure 9.11. This section presents and explains the interesting judgements. Please refer to Appendix B for an exhaustive listing.

The various judgement forms are interdependent. We present them in a top-down fashion starting from how machine states are checked and proceeding down until we present the typing rules for values. Figure 9.12 contains the rules for type checking machine states and heaps.

The type-checking rule for machine states captures the difference between the heap and register file. The type of the heap (Ψ) only grows as the program executes. Therefore it can be used recursively in the judgement. This recursive knot-tying condition $\cdot; \Psi \vdash H : \Psi$ permits the creation of loops and cycles in the heap. In contrast, the register file contains locations whose type may change. Therefore it may not assume anything about its own type, only that H has type Ψ. Finally the type-checking rule for machine states uses the notation for extended domains to ensure that the linear and non-linear parts of the heap do not overlap.

The rules for heaps check that each sub-heap is well-typed. The rule (H-union) ensures that different parts of the heap do not overlap. A type environment Δ is needed for this judgement form because of templates. When checking a template $\texttt{tmpl}[\rho]\Psi_1.H$, ρ is in scope inside H. Registers are distinguished from arbitrary locations because they may only contain word-sized values. Other locations may contain large values such as templates and code blocks.

Figure 9.13 shows the judgements for typing heap values. The block making up a code block is type-checked assuming the code's pre-condition has been met. The interesting rule is that for templates.

The template rule (h-tmpl) is designed to ensure that the template is position

$$\boxed{\vdash (H, R, I)}$$

$$\frac{\mathrm{dom}^+ H \cap \mathrm{dom}^+ R = \emptyset \qquad \begin{array}{c} \cdot \vdash \Psi \qquad \cdot \vdash \Gamma \\ \cdot; \Psi \vdash H : \Psi \end{array} \qquad \cdot; \Psi \vdash R : \Gamma \qquad \cdot; \Psi; \Gamma \vdash I}{\vdash (H, R, I)}$$

$$\boxed{\Delta; \Psi \vdash H : \Psi'}$$

$$(\text{H-empty})\frac{}{\Delta; \Psi \vdash \emptyset : \emptyset}$$

$$(\text{H-mem})\frac{\Delta; \Psi \vdash h : \sigma}{\Delta; \Psi \vdash \{l \mapsto h\} : \{l \mapsto \sigma\}}$$

$$(\text{H-reg})\frac{\Delta; \Psi \vdash w : \tau}{\Delta; \Psi \vdash \{r' \mapsto w\} : \{r' \mapsto \tau\}}(r' \in r, \mathtt{pc})$$

$$(\text{H-union})\frac{\Delta; \Psi \vdash H_1 : \Psi_1 \qquad \Delta; \Psi \vdash H_2 : \Psi_2 \qquad \mathrm{dom}^+ H_1 \cap \mathrm{dom}^+ H_2 = \emptyset}{\Delta; \Psi \vdash \{H_1, H_2\} : \{\Psi_1, \Psi_2\}}$$

$$(\text{H-sub})\frac{\Delta; \Psi \vdash H : \Psi' \qquad \Delta \vdash \Psi' \leq \Psi''}{\Delta; \Psi \vdash H : \Psi''}$$

Figure 9.12: Type-checking rules for machine states M and heaps H

$$\boxed{\Delta; \Psi \vdash h : \sigma}$$

$$(\text{h-code})\frac{\Delta, \Delta'; \Psi; \Gamma \vdash I}{\Delta; \Psi \vdash \mathtt{code}[\Delta']\Gamma.I : \forall[\Delta'].\Gamma}(\Delta \cap \Delta' = \emptyset)$$

$$(\text{h-tmpl})\frac{\begin{array}{c} \Delta, \rho; \{\Psi, \Psi_1 + \rho, \Psi_2 + \rho\} \vdash H : \{\Psi_1', \Psi_2\} \qquad \Delta \vdash \{\Psi_1, \Psi_2\} \\ \mathrm{dom}^+ H = \{i | 0 \leq i \leq k\} \qquad \mathrm{dom}\,\Psi_1 = \mathrm{dom}\Psi_1' \end{array}}{\Delta; \Psi \vdash \mathtt{tmpl}[\rho = l]\Psi_1.H : [\rho](\Psi_1 \Rightarrow \Psi_2)}(\rho \notin \Delta)$$

$$(\text{h-w})\frac{\Delta; \Psi \vdash w : \tau}{\Delta; \Psi \vdash w : \tau} \qquad (\text{h-sub})\frac{\Delta; \Psi \vdash h : \sigma \qquad \Delta \vdash \sigma \leq \sigma'}{\Delta; \Psi \vdash h : \sigma'}$$

Figure 9.13: Type-checking rules for heap values h

independent and contiguous. That the template consists of contiguous memory is checked by the condition $\mathrm{dom}^{+}H = \{i|0 \leq i \leq k\}$. Position independence is harder to check. The type of holes to be filled in the template is Ψ_1. The body of the template H is type-checked assuming that it has been placed at some arbitrary location ρ and all holes have been filled. This assumption is valid because the body of the template will only become executable after all the holes have been filled. The location ρ does not equal any other location in the heap because ρ is abstract. Recall that to ensure position independence we must guarantee that there are no absolute jumps within the template. Because ρ is abstract programs cannot manufacture any pointer whose type mentions ρ and therefore cannot make an absolute jump within the template. When type-checking H, the rule assumes that the holes have been filled. The condition $\mathrm{dom}\Psi_1 = \mathrm{dom}\Psi_1'$ guarantees that the hole locations actually make up part of the template. The value l in the template is simply an integer representing the current location of the template and is not used for type-checking.

The rules for values v and w, shown in Figure 9.14, are for the most part uninteresting. One fact to note is that many of these rules are non-deterministic. Technically, Γ or even Ψ could be true multi-sets. Therefore $\Gamma(\eta)$ may not be uniquely defined. A term type-checks if there is one valid choice for which it type-checks. This flexibility is required in the proof (specifically by the Substitution Lemma). In practice, a verifier would reject programs where either Γ or Ψ were not partial functions. Two interesting rules are (w-ptr) and (v-hole).

The (w-ptr) rule allows the creation of relative pointers but also checks that the pre-condition on a code block for pc is valid. The latter is difficult.

Intuitively, the pre-condition for a block must state that the pc points to the location where the block is located. This condition could be easily verified when type-checking heaps by simply checking that if $l \mapsto \forall[\Delta]\Gamma.I$ then $\Gamma(\mathsf{pc}) = S(l)$. However, if l shows up in the type of the code block than every consumer of this value must know that it is located at l. That requirement is unacceptable as it limits the kinds of programs that can be written. Templates, too, complicate the situation by making the precise location of the code blocks they contain uncertain.

The solution we adopt is to allow the programmer to eliminate the requirement on pc at the point where a pointer to that code block is created. This purpose is accomplished by the function $\sigma@\gamma = \sigma'$. This solution is not ideal in that code types must mention the pc before being converted into pointers even though this information is always redundant.

The (v-hole) rule checks the current pc, and reads the value at the next location. Recall that when modelling instruction sizes we gave instructions containing holes size 2. The second location is for the value to be read from the hole.

The rules for blocks (Figure 9.15) symbolically simulate the execution of the block. Each instruction is checked individually. After an ordinary instruction executes the pc is symbolically incremented in the type-checking rule. The relative and absolute jump instructions both require that the pre-condition of the destination be a super-type of the current linear part of the heap without the pc register.

$\boxed{\Delta; \Psi \vdash w : \tau}$

$$\text{(w-int)}\frac{}{\Delta; \Psi \vdash i : \mathit{int}} \qquad \text{(w-sing)}\frac{}{\Delta; \Psi \vdash \mathtt{S}(i) : S(i)}$$

$$\text{(w-ptr)}\frac{\Delta \vdash \Psi(\gamma + i) \leq \sigma \quad \sigma@(\gamma + i) = \sigma'}{\Delta; \Psi \vdash \mathtt{rel}[\gamma] * i : \mathit{rel}[\gamma] \ * \sigma'}$$

$$\text{(w-inst)}\frac{\Delta; \Psi \vdash w : \mathit{rel}[\gamma] \ * \forall[\delta', \Delta'].\Gamma \qquad \Delta \vdash \gamma'}{\Delta; \Psi \vdash w[\gamma'] : \mathit{rel}[\gamma] \ * \forall[\Delta'].\Gamma(\delta' \leftarrow \gamma')}$$

$$\text{(w-sub)}\frac{\Delta; \Psi \vdash w : \tau \qquad \Delta \vdash \tau \leq \tau'}{\Delta; \Psi \vdash w : \tau'}$$

$\boxed{\Delta; \Psi; \Gamma \vdash v : \tau}$

$$\text{(v-reg)}\frac{\Delta \vdash \Gamma(r) \leq \tau}{\Delta; \Psi; \Gamma \vdash r : \tau} \qquad \text{(v-w)}\frac{\Delta; \Psi \vdash w : \tau}{\Delta; \Psi; \Gamma \vdash w : \tau}$$

$$\text{(v-hole)}\frac{\Gamma(\mathtt{pc}) = S(\gamma) \qquad \Delta \vdash \Psi(\gamma + 1) \leq \tau}{\Delta; \Psi; \Gamma \vdash [] : \tau}$$

$$\text{(v-inst)}\frac{\Delta; \Psi; \Gamma \vdash v : \mathit{rel}[\gamma] \ * \forall[\delta', \Delta'].\Gamma \qquad \Delta \vdash \gamma'}{\Delta; \Psi; \Gamma \vdash v[\gamma'] : \mathit{rel}[\gamma] \ * \forall[\Delta'].\Gamma(\delta' \leftarrow \gamma')}$$

$$\text{(v-sub)}\frac{\Delta; \Psi \vdash v : \tau \qquad \Delta \vdash \tau \leq \tau'}{\Delta; \Psi \vdash v : \tau'}$$

Figure 9.14: Type-checking rules for values v and w.

$\boxed{\Delta; \Psi; \Gamma \vdash I}$

$$\text{(I-inst)}\frac{\begin{array}{c}\Gamma(\mathtt{pc}) = S(\gamma) \qquad \{\Delta; \Psi; \Gamma\} \ \iota \ \{\Delta'; \Psi'; \Gamma'\} \\ \Delta'; \Psi'; \Gamma'[\mathtt{pc} = S(\gamma + |\iota|)] \vdash I\end{array}}{\Delta; \Psi; \Gamma \vdash \iota; I}$$

$$\text{(I-jmp)}\frac{\Gamma(\mathtt{pc}) = S(\gamma) \qquad \Delta; \Psi; \Gamma \vdash v : *\forall[].\Gamma' \qquad \Delta \vdash \Gamma \backslash \mathtt{pc} \leq \Gamma'}{\Delta; \Psi; \Gamma \vdash \mathtt{jmp} \ v}$$

$$\text{(I-jmpr)}\frac{\Gamma(\mathtt{pc}) = S(\gamma) \qquad \Delta; \Psi; \Gamma \vdash v : \mathit{rel}[\gamma + 1] \ * (\forall[].\Gamma') \qquad \Delta \vdash \Gamma \backslash \mathtt{pc} \leq \Gamma'}{\Delta; \Psi; \Gamma \vdash \mathtt{jmpr} \ v}$$

Figure 9.15: Type-checking rules for blocks

$\boxed{\{\Delta;\Psi;\Gamma\}\ \iota\ \{\Delta';\Psi';\Gamma'\}}$ (no RTCG)

$$\frac{\Gamma(r_2) = \mathit{int} \qquad \Delta;\Psi \vdash v : \mathit{int}}{\{\Delta;\Psi;\Gamma\}\ \texttt{add}\ r_1, r_2, v\ \{\Delta;\Psi;\Gamma[r_1 = \mathit{int}]\}}$$

$$\frac{\Gamma(r_2) = S(\gamma_1) \qquad \Delta;\Psi;\Gamma \vdash v : S(\gamma_2)}{\{\Delta;\Psi;\Gamma\}\ \texttt{add}\ r_1, r_2, v\ \{\Delta;\Psi;\Gamma[r_1 = S(\gamma_1 + \gamma_2)]\}}$$

$$\frac{\Gamma(r) = \mathit{int} \qquad \Delta;\Psi \vdash v : *\forall[].\Gamma' \qquad \Delta \vdash \Gamma\backslash\texttt{pc} \leq \Gamma'}{\{\Delta;\Psi;\Gamma\}\ \texttt{bnz}\ r, v\ \{\Delta;\Psi;\Gamma\}}$$

$$\frac{\Delta;\Psi \vdash v : \tau}{\{\Delta;\Psi;\Gamma\}\ \texttt{mov}\ r, v\ \{\Delta;\Psi;\Gamma[r = \tau]\}}$$

$$\frac{\begin{array}{c}\Delta;\Psi;\Gamma \vdash v : S(\gamma) \\ \Psi(\gamma) = \sigma \qquad \sigma@\gamma = \sigma'\end{array}}{\{\Delta;\Psi;\Gamma\}\ \texttt{forget}\ r, v\ \{\Delta;\Psi;\Gamma[r = *\sigma']\}}$$

Figure 9.16: Type-checking instructions ι (no RTCG).

The subtyping relationship will hold only if the destination too has no requirement on the `pc` register. But type-checking code always places a requirement on the `pc`. The only place that the `pc` requirement can be eliminated is in rule (w-ptr). Therefore, code with no explicit requirement for `pc` implicitly will satisfy the pre-condition after the jump.

Figure 9.16 shows the rules for the non-rtcg instructions and Figure 9.17 shows the remaining instructions. In each case the rules symbolically simulate the effect of the instruction in a straightforward manner. Please refer to Section 9.1.1 and Section 9.2.3 for a detailed discussion.

The subtyping relationship referred to in the previous rules is defined in Figure 9.18. This relationship is intentionally simple. It permits heap types to drop locations, introduces a super-type *junk* for word-sized types, and allows template types to drop locations. Each of these facets is needed to type check useful code. Without a super-type for word-sized values pre-conditions for code blocks would have to specify a precise type for each register. Without being able to drop locations from template types, generated code could only be composed of a statically known number of templates. Richer, more traditional kinds of subtyping could be added to the language without difficulty, albeit at the cost of a slightly more complicated proof of soundness.

$\boxed{\{\Delta;\Psi;\Gamma\}\ \iota\ \{\Delta';\Psi';\Gamma'\}}$ (RTCG)

$$\frac{\Gamma' = \{\Gamma, \rho \mapsto [\rho'].\emptyset \Rightarrow \emptyset\}}{\{\Delta;\Psi;\Gamma\}\ \texttt{start}\ [\rho]r\ \{\Delta,\rho;\Psi;\Gamma'[r = S(\rho)]\}}(\rho \notin \Delta)$$

$$\frac{\Gamma(r) = S(\gamma) \qquad \Gamma = \{\Gamma', \gamma \mapsto [\rho'].(\emptyset \Rightarrow \Psi')\}}{\{\Delta;\Psi;\Gamma\}\ \texttt{end}\ [\rho]r\ \{\Delta,\rho;\{\Psi,\Psi' + \rho(\rho' \leftarrow \rho)\};\Gamma'[r = S(\rho)]\}}(\rho \notin \Delta)$$

$$\frac{\begin{array}{c}\Gamma(r_1) = S(\gamma) \quad \Gamma = \{\Gamma', \gamma \mapsto [\rho''].(\Psi_1 \Rightarrow \Psi_2)\} \quad \Gamma(v) = *[\rho'].(\Psi'_1 \Rightarrow \Psi'_2)\\ \sigma = [\rho''].(\{\Psi_1, \Psi'_1 + \delta(\rho' \leftarrow \rho'' + \delta)\} \Rightarrow \{\Psi_2, \Psi'_2 + \delta(\rho' \leftarrow \rho'' + \delta)\})\end{array}}{\{\Delta;\Psi;\Gamma\}\ \texttt{copy}\ [\delta]r_1, r_2, v\ \{\Delta,\delta;\Psi;\{\Gamma', \gamma \mapsto \sigma\}[r_2 = S(\delta)]\}}(\delta \notin \Delta)$$

$$\frac{\begin{array}{c}\Gamma(r_1) = S(\gamma_1) \qquad \Gamma(r_2) = S(\gamma_2) \qquad \Delta;\Psi;\Gamma \vdash v : \tau\\ \Gamma = \{\Gamma', \gamma_1 \mapsto [\rho'].\{\Psi', \gamma_2 \mapsto \tau\} \Rightarrow \Psi''\}\end{array}}{\{\Delta;\Psi;\Gamma\}\ \texttt{fill}\ r_1, r_2, v\ \{\Delta;\Psi;\{\Gamma', \gamma_1 \mapsto [\rho'].\Psi' \Rightarrow \Psi''\}\}}(\rho' \notin \Delta)$$

$$\frac{\begin{array}{c}\Gamma(r_1) = S(\gamma_1) \qquad \Gamma(r_2) = S(\gamma_2) \qquad \Gamma(r_3) = S(\gamma_3)\\ \Psi''(\gamma_3) = \sigma \qquad \sigma @ \rho' + \gamma_3 = \sigma'\\ \Gamma = \{\Gamma', \gamma_1 \mapsto [\rho'].(\{\Psi', \gamma_2 \mapsto \mathit{rel}[\rho' + \gamma_2]\ * \sigma'\} \Rightarrow \Psi'')\end{array}}{\{\Delta;\Psi;\Gamma\}\ \texttt{fillr}\ r_1, r_2, r_3\ \{\Delta;\Psi;\{\Gamma', \gamma_1 \mapsto [\rho'].(\Psi' \Rightarrow \Psi'')\}\}}(\rho' \notin \Delta)$$

Figure 9.17: Type-checking instructions ι (RTCG).

$\boxed{\Delta \vdash \Psi_1 \leq \Psi_2,\ \Delta \vdash \sigma \leq \sigma'}$

$$\frac{}{\Delta \vdash \sigma \leq \sigma} \qquad \frac{}{\Delta \vdash \tau \leq \mathit{junk}}$$

$$\frac{\Delta \vdash \Psi_2 \leq \Psi'_2}{\Delta \vdash [\rho]\Psi_1 \Rightarrow \Psi_2 \leq [\rho]\Psi_1 \Rightarrow \Psi'_2}$$

$$\frac{\Delta \vdash \sigma_1 \leq \sigma'_1 \quad \ldots \quad \Delta \vdash \sigma_k \leq \sigma'_k}{\Delta \vdash \{\eta_1 \mapsto \sigma_1, \ldots, \eta_n \mapsto \sigma_n\} \leq \{\eta_1 \mapsto \sigma'_1, \ldots, \eta_k \mapsto \sigma'_k\}}(0 \leq k \leq n)$$

Figure 9.18: TAL/T subtyping

9.4 The Implementation

The formalism presented here reflects lessons learned since we implemented TAL/T. In the implementation, templates are copied into special buffers called code regions which have their own typing rules. Our formalism shows that there is no need for this distinction; templates suffice.

Originally our implementation did not track the difference between relative and absolute jumps and was therefore unsound. When we discovered this flaw, we fixed the implementation by distinguishing absolute addresses from relative addresses. Still, the implementation treats labels abstractly making it hard to formalize this distinction. By making locations integers, our model is more faithful to the underlying machine than the original TAL.

Our early work on alias types inspired their addition to the TAL implementation. As such they are treated orthogonally from the rest. Our formalism suggests that they can be smoothly integrated with the whole heap thus unifying the mechanisms used for heap locations and registers. This simplification would permit programs to store linear parts of the heap in their static data areas. This capability can be useful for simulating extra registers or initializing data. We believe that the model presented here can be easily extended to support the full polymorphism of alias types.

On the other hand our model lacks many key features; such as dynamic allocation, tuples, a stack, precise control over data layouts, a module system; that are an integral part of TAL/T. These features are largely orthogonal to those presented here, and therefore could be integrated without invalidating our proof of soundness.

One important feature that we do not model is memory reclamation. Our model supposes an infinite amount of memory. In practice, the implementation relies on a garbage collector to reclaim memory automatically without programmer intervention. However, garbage collectors place stringent requirements on the ways programs can manufacture pointers. At the very least, programs should only create pointers to an object from pointers into that object.

Because our language supports arithmetic on singleton integers, it is possible to violate this condition. For example the singletons $\mathtt{S}(i-j)$ and $\mathtt{S}(j)$ can be added to produce a pointer to the location i. The garbage collector may have reclaimed this memory causing the program to possibly crash. Because the abstract machine does not model the garbage collector, our soundness proof does not establish this crucial property.

Informally, this problem is easily solved. Simply distinguish singleton pointers from singleton integers and restrict arithmetic on singleton pointers appropriately. This choice is easy to implement and suffices to guarantee soundness with respect to the collector. Proving this formally would be somewhat challenging.

Finally, our model does not capture fall-through. In the implementation, control may be implicitly transfered from one template to another by terminating a template with a partial block (one not terminated with a jump). The same functionality can be expressed in our formal model using a hole in a relative jump instruction at the cost of an additional jump. We chose not to model this feature to keep the type system clean. To model it requires adding a post-condition to template types, and a pseudo-instruction to instantiate the type variables on the fall through. These changes are not technically difficult.

Chapter 10

Conclusion & Future Work

This thesis has demonstrated efficient certified run-time code generation. Our system guarantees, based on compile-time checks alone, that well-typed programs, even those using RTCG, will not crash. This strong guarantee is obtained by using an optimizing and certifying compiler relying on a template-based approach to RTCG. This approach permits fast code generation but limits the ways in which dynamic code can be assembled.

Our evaluation shows that for some kinds of applications this approach yields good performance provided standard control-flow based optimizations are applied to the templates at compile time. To this effect, we have shown how to adapt standard optimizations for template-based RTCG. Finally, we have formally shown that the output language of our compiler TAL/T, Typed Assembly Language extended with Templates, is sound.

These achievements suggest a number of interesting direction to pursue.

Cyclone. Our source language Cyclone is simple to use requiring no new types and only four new RTCG related constructs: `codegen`, `cut`, `splice`, and `fill`. The cost of this simplicity is that programmers must use function calls to create certain kinds of control flow in dynamically generated code. An alternative design might introduce explicit code types. Functions could pass and return code fragments to and from other functions.

Type systems supporting this level of flexibility are well understood [Dav96, DP96, TBS98, Tah99a]. Languages like MetaML make use of them. These techniques carry over into a low-level C-like language such as Cyclone but the complex term structure of the language would likely make the types significantly larger and thus less palatable to progrmmers.

A language providing more flexibility for generating code would benefit programmers but would be harder to implement. In such a language, dynamic code fragments may be used in a many different contexts making traditional optimizations applied at compile time less precise. A more expressive language may therefore require heavyweight RTCG to achieve high performance.

TAL/T. The assembly language we prove sound yields a very strong guarantee and permits many interesting programs to be written. For example, an extension to Cyclone as described above would likely be easily supported. However, there are at least three significant ways in which TAL/T could be improved.

1. TAL/T could have principle types for instructions.

   A template containing a single instruction can be given many different types. A principle type is the unique type that subsumes all the other types for that template. If instructions had principle types than TAL/T could emit individual instructions rather than copying whole blocks of code. Extending the type system to accomodate such expressive types would be a significant and fundamental contribution.

2. TAL/T's type system could be simplified and lifted to the user level.

   Although TAL/T is only intended to be machine generated, a language that is hard to write is hard to understand. A simplified type system might permit some of TAL/T's powerful typing mechanisms to be lifted into a source language. Such advance typing mechanisms could help programmers express richer invariants in their types and therefore catch more bugs.

   Currently, these mechanisms are too complicated for human use. Projects like Vault [DF] and the next generation of Cyclone [JMG$^+$] are already working on developing techniques to simplify these mechanisms and so make them available to programmers. We believe that this will be a fruitful avenue of future research.

3. TAL/T's proof is for an abstract machine rather than the actual machine our system is running on. We rely on the reader believing a meta-theorem that our proof is "good enough." Clearly this situation could be improved.

   Already significant progress has been made in this direction [App01, AF00]. The extension to real machines makes the proofs significantly more complex and therefore harder to check. To address this concern, that research has relied on mechanically checked proofs. It would be interesting to extend those mechanical proofs to support RTCG.

   Our experience suggests that encoding such proofs in theorem provers is hard. We would like to see projects aimed at producing theorem provers (or even proof checkers) capable of concisely and directly representing proofs about semantics. One hopeful project with this explicit goal is Twelf [PS99a, PS98] although we were unable to encode our soundness proof within it.

Optimization. One contribution of this thesis has been to show how Cyclone can be compiled, and how standard analyses and optimizations can be applied to dynamically generated functions. This capability is crucially important for achieving the performance benefits of RTCG. There are a number of ways to extend our work.

Most optimizations and analyses extend without change to templates. We might call them *template-invariant* if correctness in a standard language without templates, implies correctness in a language with templates. Our intutition suggests that only optimizations involving code motion are not template-invarant. It would be useful to ground this intuition by formally defining and proving the conditions for template-invariance.

Another interesting research question is the question of how to compile a language with richer kinds of RTCG. For example, we might wish to compile the extension to Cyclone supporting first-class code described above. We believe that our analysis techniques extended inter-procedurally would yield correct results but do not know whether such an approach would be practical. There may be too little context in such a setting to effectively optimize dynamically-generated code at compile time.

Evaluation. Clearly, more work to evaluate RTCG must be done. The lack of compelling example applications is particularly telling. We have presented two larger applications and several micro-benchmarks similar to those in previous work. Through a detailed analysis, we have shown that 1) staying within the instruction cache is crucial for performance, 2) algorithmic alternatives sometimes provide the same gains as RTCG, 3) data specialization is often viable, and 4) RTCG can be very sensitive to input parameters. Despite these discouraging results, RTCG does, in many of the cases we tried, significantly accelerate programs.

Establishing that a large class of applications would benefit from RTCG requires a much broader evaluation. Because our system uses our own programming language, porting example applications is time consuming. Most other systems have presented some performance numbers with applications similar to our own. Initial steps towards a comprehensive study have been taken using DyC [GMP$^+$00]. Unfortunately, most published results have focused solely on break-even points and the cost of code generation. Those analyses have not examined the causes behind the observed performance, or explored alternative algorithmic approaches for improving performance.

Furthermore, all RTCG research has been performed by the developers of their respective systems. Being highly committed to RTCG and having spent hundreds of hours developing a system, in my experience, makes scientific and objective evaluations hard to make. It would be best if a disinterested third party took an existing system for RTCG and dispassionately examined the costs and benefits of using it for a class of large well-optimized applications (for example, the SPEC benchmarks).

A second question we would like to see answered is: For what applications is lightweight RTCG preferable to heavyweight RTCG? Computers today have large physical memories and are very fast. Including the compiler with the program's executable is not unthinkable. Because processors are so quick, functions are often called tens of thousands of times. Under these conditions, the cost of heavyweight compilation might be easily recovered. The only work in this area was done for 'C. That work concluded that the choice between heavyweight and lightweight RTCG depended on the application [PHEK99]. It would be useful to identify applications belonging in each of these classes.

Bibliography

[Ado85] Adobe Systems Incorporated, Reading, MA, USA. *PostScript Language Reference Manual*, 1985.

[AF00] Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *27th ACM Symposium on Principles of Programming Languages*, pages 243–253, January 2000.

[And95] Peter Holst Andersen. Partial evaluation applied to ray tracing. Technical Report 95/2, DIKU, University of Copenhagen, January 1995.

[APC+96] 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.

[App01] Andrew W. Appel. Foundational proof-carrying code. In *16th Annual IEEE Symposium on Logic in Computer Science*, 2001. To appear.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques and Tools*. Addison-Wesley, 1986.

[BAB96] Doug Burger, Todd M. Austin, and Steve Bennett. Evaluating future microprocessors: The simplescalar tool set. Technical Report CS-TR-1996-1308, University of Wisconsin-Madison, Computer Sciences Department, 1996.

[BBB+91] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, 1991.

[BCF+99] Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, Vugranam C. Sreedhar, Harini Srinivasan, and John Whaley. The jalapeno dynamic optimizing compiler for java. In *ACM Java Grande Conference*, pages 129–141, 1999.

[Bru72] N. G. De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation. In *Indag. Math.*, volume 34, pages 381–392, 1972.

[Bur98] Doug Burger. *Hardware Techniques to Improve the Performance of the Processor/Memory Interface*. Ph.D. dissertation, University of Wisconsin-Madison, December 1998.

[CD93] Charles Consel and Olivier Danvy. Partial evaluation: Principles and perspectives. *Jouneés Francophones des Langages Applicatifs*, 1–2, February 1993.

[CHN+96] C. Consel, L. Hornof, F. Noel, J. Noy'e, and E. Volanschi. A uniform approach for compile-time and run-time specialization. In O. Danvy, R. Gluck, and P. Thiemann, editors, *Lecture Notes in Computer Science*, volume 1110, pages 54–72, 1996.

[CK94] Bob Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. *ACM SIGMETRICS Performance Evaluation Review*, 22(1):128–137, May 1994.

[CN96] Charles Consel and François Noë. A general approach for run-time specialization and its application to c. In *23rd ACM Symposium on Principles of Programming Languages*, pages 145–170, January 1996.

[Cor97] Intel Corporation. *Intel Architecture Optimization Manual*. Intel Corporation, P.O. Box 7641, Mt. Prospect, IL, 60056-7641, 1997.

[CW00] Karl Crary and Stephanie Weirich. Resource bound certification. In *Twenty-Seventh ACM Symposium on Principles of Programming Languages*, pages 184–198, Boston, MA, January 2000.

[Dav96] Rowan Davies. A temporal-logic approach to binding-time analysis. In *Proceedings of the 11th annual Symposium on Logic in Computer Science*, pages 184–195, July 1996.

[DF] R. DeLine and Manuel Fahndrich. The vault project. See website at http://wwww.research.microsoft.com/projects/Vault.

[DHa] Damien Doligez and Robert Harley. Fractal. GML source is available from http://caml.inria.fr/icfp00-contest.

[DHb] Damien Doligez and Robert Harley. Go. GML source is available from http://caml.inria.fr/icfp00-contest.

[DHc] Damien Doligez and Robert Harley. Kal. GML source is available from http://caml.inria.fr/icfp00-contest.

[DP96] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In *23rd ACM Symposium on Principles of Programming Languages*, pages 258–270, January 1996.

[EHK96] Dawson Engler, Wilson Hsieh, and M. Frans Kaashoek. 'C: A language for fast, efficient, high-level dynamic code generation. In *23rd ACM Symposium on Principles of Programming Languages*, pages 131–144, January 1996.

[FB99] B. Fisk and I. Bahar. The non-critical buffer: Using load latency tolerance to improve data cache efficiency. In *IEEE Conference on Computer Design*, October 1999.

[Fel87] Matthias Felleisen. *The calculi of lambda-v-cs-conversion: A syntactic theory of control and state in imperative higher-order programming languages.* Ph.D. dissertation, Indiana University, 1987.

[Gir71] Jean-Yves Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination de coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 63–92. North-Holland Publishing Co., 1971.

[GKM82] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a call graph execution profiler. In *Proceedings of the SIGPlan '82 Symposium on Compiler Construction*, pages 120–126, Boston, MA, 1982.

[Gle99] Neal Glew. Type dispatch for named hierarchical types. In *ACM International Conference on Functional Programming*, pages 172–182, Paris, France, September 1999.

[GM99] Neal Glew and Greg Morrisett. Type safe linking and modular assembly language. In *26th ACM Symposium on Principles of Programming Languages*, San Antonio, Texas, USA, January 1999. To appear.

[GM00] Dan Grossman and Greg Morrisett. Scalable certification of native code: Experience from compiling to TALx86. Technical Report TR2000-1783, Cornell University, February 2000.

[GMP+97] 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, 1997. Updated May 12, 1999.

[GMP+00] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. DyC: An expressive annotation-directed dynamic compiler for C. *ACM Transactions on Progamming Languages and Systems*, 248(1-2):147–100, October 2000.

[GNZ95] Robert Glück, Ryo Nakashige, and Robert Zöchling. Binding-time analysis applied to mathematical algorithms. In J. Doležal and J. Fidler, editors, *System Modelling and Optimization*, pages 137–146. Chapman & Hall, 1995.

[GPM+99] 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.

[HJ99] 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.

[IO01] Samin Ishtiaq and Peter O'Hearn. Bi as an assertion language for mutable data structures. In *28th ACM Symposium on Principles of Programming Languages*, pages 14–26, New York, 2001.

[JGS93] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[JMG+] Trevor Jim, Greg Morrisett, Dan Grossman, Mike Hicks, Mathieu Baudet, Matthew Harris, and Yanling Wang. The Cyclone project. See the website at http://www.cs.cornell.edu/projects/cyclone.

[KEH91] David Keppel, Susan J. Eggers, and Robert R. Henry. A case for runtime code generation. Technical Report 91-11-04, Department of Computer Science and Engineering, University of Washington, November 1991.

[Kis97] Thomas Kistler. Dynamic runtime optimization. In *Proceedings of the Joint Modular Languages Conference, JMLC*, March 1997. Also published as Springer Lecture Notes in Computer Science volume 1204.

[Kis99] Thomas Kistler. *Continuous Program Optimization*. Ph.D. dissertation, University of California, Irvine, November 1999.

[KKR+86] David Kranz, R. Kelsey, J. Rees, P. R. Hudak, J. Philbin, and N. Adams. ORBIT: An optimizing compiler for Scheme. In *Proceedings of the ACM '86 Symposium on Compiler Construction*, pages 219–233, June 1986.

[Koz98] Dexter Kozen. Efficient code certification. Technical Report 98-1661, Department of Computer Science, Cornell University, 4130 Upson Hall, Ithaca, NY 14853-7501, USA, January 1998.

[Koz99] Dexter Kozen. Language-based security. In *Proccedings of the Conference on Mathematical Foundations of Computer Science*, volume 1672, pages 284–298. Springer-Verlag, 1999.

[KR] Leif Kornstaedt and Andreas Rossberg. Chess. GML source is available from http://www.ps.uni-sb.de/helikopter.

[KR88] B. W. Kernighan and D. M. Ritchie. *The C Programming Language, Second Edition.* Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[KR96] Todd Knoblock and Eric Ruf. Data specialization. In *ACM Conference on Programming Language Design and Implementation*, pages 215–255, May 1996.

[Lan64] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1964.

[Ler92] Xavier Leroy. Unboxed objects and polymorphic typing. In *19th ACM Symposium on Principles of Programming Languages*, pages 177–188, Albuquerque, January 1992.

[LFK+93] P. Geoffrey Lowney, Stefan M. Freudenberger, Thomas J. Karzes, W. D. Lichtenstein, Robert P. Nix, John S. O'Donnell, and John C. Ruttenberg. The Multiflow Trace Scheduling compiler. *The Journal of Supercomputing*, 7(1-2):51–142, 1993.

[LL94] 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.

[LL96] 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.

[LL98] Mark Leone and Peter Lee. Dynamic specialization in the Fabius system. *ACM Computing Surveys*, 30(3es):??–??, 1998.

[LRYT99] Alvin R. Lebeck, David R. Raymond, Chia-Lin Yang, and Mithuna Thottethodi. Annotated memory references: A mechanism for informed cache management. In *European Conference on Parallel Processing*, pages 1251–1254, 1999.

[MCG+99] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. Talx86: A realistic typed assembly language. In *ACM Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, GA, May 1999.

[MCGW98] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In *Workshop on Types in Compilation*, pages 95–118, Kyoto, Japan, March 1998.

[Mic99] Sun Microsystems. The java hotspot tm server vm, 1999.

[Mog86] T. Mogensen. The application of partial evaluation to ray-tracing. Masters dissertation, University of Copenhagen, Denmark, 1986.

[Mor95] Gregory Morrisett. *Compiling with Types.* Ph.D. dissertation, Carnegie Mellon University, 1995. Published as CMU Technical Report CMU-CS-95-226.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, 1997.

[MWCG99] 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.

[Nec97] George Necula. Proof-carrying code. In *24th ACM Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, 1997.

[NHCL96] 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.

[NL96] George Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of Operating System Design and Implementation*, pages 229–243, Seattle, WA, USA, October 1996.

[NL98a] George Necula and Peter Lee. The design and implementation of a certifying compiler. In *ACM Conference on Programming Language Design and Implementation*, pages 333–344, Montreal, Canada, June 1998.

[NL98b] George Necula and Peter Lee. The design and implementation of a certifying compiler. In *ACM Conference on Programming Language Design and Implementation*, pages 333 – 344, Montreal, June 1998.

[O'H93] Peter O'Hearn. A model for syntactic control of interference. *Mathematical Structures in Computer Science*, 3(4):435–465, 1993.

[O'H00] Peter O'Hearn. On bunched typing. Unpublished manuscript, July 2000.

[PEK96] Massimiliano Poletto, Dawson Engler, and M. Frans Kaashoek. tcc: a template-based compiler for ‘C. In *ACM Workshop on Compiler Support for System Software*, pages 1–7, Tuscon, Arizona, February 1996.

[PEK97] 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.

[PFP99] Sanjay J. Patel, Daniel H. Friendly, and Yale N. Patt. Evaluation of design options for the trace cache fetch mechanism. *IEEE Transactions on Computers*, 48(2):193–204, 1999.

[PHEK99] Massimiliano Poletto, Wilson C. Hsieh, Dawson R. Engler, and M. Frans Kaashoek. ‘C and tcc: A language and compiler for dynamic code generation. *ACM Transactions on Progamming Languages and Systems*, 21(2):324–369, March 1999.

[Pol99] Massimiliano Poletto. *Language and compiler support for dynamic code generation*. Ph.D. dissertation, Massachusetts Institute of Technology, June 1999.

[PS98] Frank Pfenning and Carsten Schurmann. *Twelf User's Guide*. Carnegie-Mellon University, September 1998. Also available as technical report CMU-CS-98-173.

[PS99a] Frank Pfenning and Carsten Schurmann. Twelf - a meta-logical framework for deductive systems. In H.Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction*, volume 1632, pages 202–206, Trento, Italy, July 1999. Springer-Verlag.

[PS99b] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Progamming Languages and Systems*, 21(5):895–913, September 1999.

[RC91] Jonathan Rees and William Clinger. Revised 4 report on the algorithmic language scheme. *Lisp Pointers*, 4(3):1–55, 1991.

[Rey74] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425, New York, 1974.

[RM] John Reppy and Greg Morrisett. The third annual icfp programming contest. Available from http://www.cs.cornell.edu/icfp.

[SAMC98] Kevin Skadron, Pritpal S. Ahuja, Margaret Martonosi, and Douglas W. Clark. Improving prediction for procedure returns with return-address-stack repair mechanisms. In *International Symposium on Microarchitecture*, pages 259–271, 1998.

[Ste78] Guy Steele Jr. Rabbit: A compiler for Scheme. Masters dissertation, MIT, 1978.

[SWM99] Frederick Smith, David Walker, and Greg Morrisett. Alias types. Technical Report TR99-1773, Cornell University, October 1999.

[SWM00] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *European Symposium on Programming*, pages 366–381, Berlin, March 2000.

[Tah99a] Walid Taha. A sound reduction semantics for untyped CBN multi-stage computation. or, the theory of MetaML is non-trival. *ACM SIGPLAN Notices*, 34(11):34–43, 1999.

[Tah99b] Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. Ph.D. dissertation, Oregon Graduate Institute of Science and Technology, June 1999.

[Tah00] Walid Taha. A sound reduction semantics for untyped CBN multi-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.

[TBS98] Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. Multi-stage programming: Axiomatization and type safety. Technical Report 98-002, Oregon Graduate Institute of Science and Technology, 1998.

[TMC+96] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *ACM Conference on Programming Language Design and Implementation*, pages 181–192, Philadelphia, May 1996.

[Tra] Transmeta. Website is http://www.transmeta.com.

[TS97] 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.

[Wat92] Bill Watterson. *Attack of the Deranged Mutant Killer Snow Goons*. Andrews McMeel Publishing, April 1992.

[WC97] Andrew K. Wright and Robert Cartwright. A practical soft type system for scheme. *ACM Transactions on Progamming Languages and Systems*, 19(1):87–152, January 1997.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[Whi80] Turner Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, 1980.

[WLP98] 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.

[WLPD98] Philip Wickline, Peter Lee, Frank Pfenning, and Rowan Davies. Modal types as staging specifications for run-time code generation. In *ACM Computing Surveys*, 1998. Special issue on Partial Evaluation.

[WM00] David Walker and Greg Morrisett. Alias types for recursive data structures (extended version). Technical Report TR2000-1787, Cornell University, March 2000.

[WR96] Emmett Witchel and Mendel Rosenblum. Embra: Fast and flexible machine simulation. In *SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 68–79, 1996.

[XP98] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *ACM Conference on Programming Language Design and Implementation*, pages 249–257, Montreal, June 1998.

[XP99] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *26th ACM Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, TX, January 1999.

Appendix A

Micro-benchmark Source Code

This appendix presentes the source code to the micro-benchmarks discussed in Chaper 4.

```
void cheb(float func(float), float c[], float xa, float xb) {
  int n = size(c);
  int k, j;
  float xm, xp, sm;
  float f[] = new_array_float(n+1);

  xp = (xb + xa) / 2;
  xm = (xb − xa) / 2;

  for(k=1; k<=n; k++) {
    f[k] = func(xp + xm * cos(pi * (k − 0.5) / n));
  }

  for(j=0; j<=n−1; j++) {
    sm = 0.0;
    for(k=1; k<=n; k++) {
      sm = sm + f[k] * cos(pi * j * (k − 0.5) / n);
    }
    c[j] = (2.0 / n) * sm;
  }
}
```

Figure A.1: Cyclone source for `cheb` (no RTCG)

```
void cheb_gen(int n)(float func(float),float c[], float xa, float xb) {
  *(float) f[] = new_array(n+1,new (0.0f));

  for(int i=0;i<=n;i++)
    f[i] = ^(0.0f);

  return codegen(void cheb(float func(float), float c[],
                 float xa, float xb) {
    float xm, xp, sm = 0.0;

    xp = (xb + xa) / 2;
    xm = (xb - xa) / 2;

    cut {
      for(int k=1; k<=n; k++) {
        splice {
          fill(f[k]).1 = func(xp +
                              xm * fill(^(cos(pi*(k-0.5)/n))).1);
        }
      }
      for(int j=0; j<=n-1; j++) {
        splice { sm = 0.0; }
        for(int k=1; k<=n; k++) {
          _ v = cos(pi * j * (k - 0.5) / n);
          if(v != 0.0) {
            if      (v == 1.0) splice { sm += fill(f[k]).1; }
            else if (v == -1.0) splice { sm -= fill(f[k]).1; }
            else                splice { sm += fill(f[k]).1 *
                                              fill(^(v)).1; }
          }
        }
        splice{ c[fill(j)] = fill(^(2.0 / n)).1 * sm; }
      }
    }
  });
}
```

Figure A.2: Cyclone source for `cheb` (RTCG)

```
void csi(float x[], float y[], float z[]) {
  int n = size(x) − 1;
  float h[] = new_array_float(n+1);
  float b[] = new_array_float(n+1);
  float u[] = new_array_float(n+1);
  float v[] = new_array_float(n+1);

  int i = 0;
  for(; i<= n−1; i=i+1){
    h[i] = x[i+1]−x[i];
    b[i] = (6/h[i])*(y[i+1]−y[i]);
  }

  u[1] = 2*(h[0]+h[1]);
  v[1] = b[1]−b[0];
  for (int i = 2; i <= n−1; i=i+1){
    u[i] = 2*(h[i]+h[i−1])−h[i−1]*h[i−1]/u[i−1];
    v[i] = b[i]−b[i−1]−h[i−1]*v[i−1]/u[i−1];
  }

  z[n] = 0;

  i=n−1;
  if (i >= 1)
    do {
      z[i] = (v[i]−h[i]*z[i+1])/u[i];
      i=i−1;
    } while (i >= 1);
  z[0] = 0;
}
```

Figure A.3: Cyclone source for `csi` (no RTCG)

Figure A.4: Cyclone source for `csi` (RTCG)

```
void scramble(double fr[], double fi[], int n) {
  for(int m=1,j=0; m<n−1;m++) {
    for(int k=n>>1; ((j^=k)&k) == 0; k>>=1)
      /* Do nothing. */;
    if(j>m) {
      double tmp;
      tmp = fr[m]; fr[m] = fr[j]; fr[j] = tmp;
      tmp = fi[m]; fi[m] = fi[j]; fi[j] = tmp;
} } }

void fft(double fr[], double fi[], int ldn, int is) {
  int n2 = 1 << ldn;
  double is_pi = is * pi;

  scramble(fr,fi,n2);

  for(int ldm=1; ldm <= ldn; ldm++) {
    int m = 1 << ldm, mh = m >> 1;
    double phi = is_pi / mh, w = 0.0;
    for(int j=0; j < mh; j++) {
      w += phi;
      double c = cos(w), s = sin(w);

      for(int r=0; r<n2; r+=m) {
        int t1 = r+j, t2 = t1+mh;

        double fr2 = fr[t2], fi2=fi[t2];
        double vr = fr2 * c − fi2 * s;
        double vi = fr2 * s + fi2 * c;

        double ur = fr[t1];
        fr[t1] = ur+vr;
        fr[t2] = ur−vr;
        double ui = fi[t1];
        fi[t1]=ui+vi;
        fi[t2]=ui−vi;
} } } }
```

Figure A.5: Cyclone source for `fft` (no RTCG)

```
void fft_gen(int ldn,int is)(double fr[], double fi[]) {
  int n2 = 1 << ldn;
  double is_pi = is * pi;
  return codegen(void fft(double fr[], double fi[]) {
    cut {
      for(int m=1,j=0; m<n2−1;m++) { // scramble inlined
        for(int k=n2>>1; ((j^=k)&k) == 0; k>>=1)
          /* Do nothing. */;

        if(j>m) {
          splice {
            double tmp;
            int m = fill(m), j = fill(j);
            tmp = fr[m]; fr[m] = fr[j]; fr[j] = tmp;
            tmp = fi[m]; fi[m] = fi[j]; fi[j] = tmp;
      } } }

      for(int ldm=1; ldm <= ldn; ldm++) {
        int m  = 1 << ldm, mh = m >> 1;
        double phi = is_pi / mh, w = 0.0;

        for(int j=0; j < mh; j++, w+=phi) {
          double c = cos(w), s = sin(w);
          splice {
            double c = fill(^(c)).1, s = fill(^(s)).1;

            for(int r=0; r<fill(n2); r+=fill(m)) {
              int t1 = r+fill(j), t2 = r+fill(j+mh);
              double fr2 = fr[t2], fi2 = fi[t2];
              double vr = fr2 * c − fi2 * s;
              double vi = fr2 * s + fi2 * c;

              double ur = fr[t1];
              fr[t1] = ur+vr;
              fr[t2] = ur−vr;

              double ui = fi[t1];
              fi[t1]=ui+vi;
              fi[t2]=ui−vi;
} } } } } }); }
```

Figure A.6: Cyclone source for `fft` (RTCG)

```
float romberg(float f(float), float a, float b, int M)
{
  int n, m, i;
  float h, s;
  float r0[] = new_array_float(M+1);
  float r1[] = new_array_float(M+1);

  h = b - a;
  r0[0] = (f(a) + f(b)) * h / 2.0;

  for(n=1; n <= M; n++) {
    h = h / 2.0;
    s = 0.0;

    _ lim = 1 << n-1;
    for(i=1; i <= lim; i++)
      s = s + f(a + (:float)(2.0 * i - 1) * h);

    r1[0] = r0[0]/2.0 + h * s;
    for(m=1; m <= n; m++) {
      _ p   = (1 << (2*m)) - 1;
      r1[m] = r1[m-1] + (1.0/p) * (r1[m-1] - r0[m-1]);
    }

    _ tmp = r0; r0 = r1; r1 = tmp;
  }
  return r0[M];
}
```

Figure A.7: Cyclone source for `romberg` (no RTCG)

```
float romberg_gen(float a,float b, int M)(float f(float)) {
  *(float) r0[] = new_array(M+1,^(0.0f));
  *(float) r1[] = new_array(M+1,^(0.0f));
  float h = b − a;

  for(int i=0; i<=M;i++) {
    r0[i] = ^(0.0f); r1[i] = ^(0.0f);
  }

  return codegen(float romberg(float f(float)) {
    float a = fill(^(a)).1, b = fill(^(b)).1;

    fill(r0[0]).1 = (f(a) + f(b)) * fill(^(h/2.0)).1;

    cut {
      for(int n = 1; n <= M; n++) {
        h = h / 2.0;
        _ lim = 1 << (n−1);
        splice {
          float h = fill(^(h)).1, s = 0.0;

          for(int i = 1; i <= fill(lim); i++)
            s = s + f(a + (:float)(2.0 * i − 1) * h);
          fill(r1[0]).1 = fill(r0[0]).1/2.0 + h * s;
        }

        for(int m = 1; m <= n; m++) {
          float p = (1.0/((1 << (2*m)) − 1));
          splice {
            float p = fill(^(p)).1;
            fill(r1[m]).1 = fill(r1[m−1]).1 +
              p * (fill(r1[m−1]).1 − fill(r0[m−1]).1);
          }
        }
        _ tmp = r0; r0 = r1; r1 = tmp;
      }
    }
      return fill(r0[M]).1;
    });
}
```

Figure A.8: Cyclone source for `romberg` (RTCG)

```
int dot(int u[],int v []) {
  int len = size(u);
  int result = 0;

  for(int i = 0; i < len; ++i)
    result += u[i] * v[i];
  return result;
}
```

Figure A.9: Cyclone source for `dot` (no RTCG)

```
int dot_gen(int u[])(int []) {
  int len = size(u);

  return codegen(int dot(int v[]) {
    int result = 0;
    cut {
      for(int i = 0; i < len; ++i) {
        _ ui = u[i];
        switch ui {
        case 0: ;
        case 1: splice { result += v[fill(i)]; }
        case 2: splice { result += v[fill(i)] << 1; }
        default: splice { result += fill(ui) * v[fill(i)]; }
        }
      }
    }
    return result;
  });
}
```

Figure A.10: Cyclone source for `dot` (RTCG)

```
int poly(int p[],int x) {
  int len = size(p);
  int x_i = 1;
  int result = 0;
  for(int i=0; i<len; ++i) {
    _ ai = p[i];
    if(ai != 0) result += ai * x_i;
    x_i *= x;
  }
  return result;
}
```

Figure A.11: Cyclone source for `poly` (no RTCG)

```
int poly_gen(int p[])(int) {
  int len = size(p);
  return codegen (int poly(int x) {
    int x_i = 1;
    int result = 0;
    cut {
      for(int i=0; i<len; ++i) {
        _ ai = p[i];
        if(ai != 0) splice { result += fill(ai) * x_i; };
        splice{ x_i *= x; };
      }
    }
    return result;
  });
}
```

Figure A.12: Cyclone source for `poly` (RTCG)

Appendix B

TAL/T: Complete Definition

This chapter provides a complete definition of TAL/T. Please refer to Chapter 9 for the motivation and intuition for the design.

B.1 Syntax

B.1.1 Core Syntax

$$
\begin{array}{llcl}
\text{(integers)} & i, j, k, l & & \\
\text{(integer types)} & \gamma & ::= & \delta \mid i \mid \gamma_1 + \gamma_2 \\
\text{(location types)} & \eta & ::= & \gamma \mid r \mid \mathtt{pc} \\
\text{(type envs)} & \Delta & ::= & \delta_1, \ldots, \delta_k \\
\text{(heap types)} & \Psi, \Gamma & ::= & \{\eta_1 \mapsto \sigma_1, \ldots, \eta_n \mapsto \sigma_n\} \\
\text{(memory types)} & \sigma & ::= & \tau \mid \forall[\Delta].\Gamma \\
\text{(word types)} & \tau & ::= & \mathit{int} \mid S(\gamma) \mid \mathit{rel}[\gamma] * \sigma \mid \mathit{junk} \\
 & & & \\
\text{(locations)} & L & ::= & i \mid r \mid \mathtt{pc} \\
\text{(registers)} & r & ::= & \mathtt{r}_1 \mid \ldots \mid \mathtt{r}_k \\
\text{(heaps)} & H, R & ::= & \{L_1 \mapsto h_1, \ldots, L_n \mapsto h_n\} \\
\text{(heap values)} & h & ::= & w \mid \mathtt{code}[\Delta]\Gamma.I \\
\text{(word values)} & w & ::= & i \mid \mathtt{S}(i) \mid \mathtt{rel}[\gamma] * i \mid w[\gamma] \\
\text{(operands)} & v & ::= & r \mid w \mid [] \mid v[\gamma] \\
\text{(instructions)} & \iota & ::= & \mathtt{add}\ r_1, r_2, v \mid \mathtt{bnz}\ r, v \mid \mathtt{mov}\ r, v \mid \\
 & & & \mathtt{forget}\ r_1, r_2, v \\
\text{(blocks)} & I & ::= & \iota; I \mid \mathtt{jmp}\ v \mid \mathtt{jmpr}\ v \\
\text{(machine states)} & M & ::= & (H, R, I)
\end{array}
$$

B.1.2 RTCG Additions

(memory types)	σ	$::=$	$\ldots \mid [\rho].\Psi_1 \Rightarrow \Psi_2$
(heap values)	h	$::=$	$\ldots \mid \mathtt{tmpl}[\rho = i]\Psi.H$
(instructions)	ι	$::=$	$\ldots \mid \ \mathtt{start}\ [\rho]r \mid \ \mathtt{end}\ [\rho]r \mid \ \mathtt{copy}\ [\delta]r_1, r_2, v \mid$
			$\mathtt{fill}\ r_1, r_2, v \mid \ \mathtt{fillr}\ r_1, r_2, r_3 \mid$

B.2 Operational Semantics

$\boxed{M \longmapsto M'}$ (no RTCG)

For the non-rtcg instructions where $(H, R, \iota; I) \longmapsto (H', R'[\mathtt{pc} = \mathtt{S}(i + |\iota|)], I')$ where $R(\mathtt{pc}) = \mathtt{S}(i)$:

$\iota =$	$H' =$	$R' =$	$I' =$
$\mathtt{add}\ r_1, r_2, v$	H	$R[r_1 = R(r_2) + S_{H,R}(v)]$	I
$\mathtt{add}\ r_1, r_2, v$	H	$R[r_1 = \mathtt{S}(i + j)]$	I
when $R(r_2) = \mathtt{S}(i)$ and $S_{H,R}(v) = \mathtt{S}(j)$			
$\mathtt{bnz}\ r, v$	H	R	I
when $R(r) = 0$			
$\mathtt{mov}\ r, v$	H	$R[r = S_{H,R}(v)]$	I
$\mathtt{forget}\ r, v$	H	$R[r = *i]$	I
when $S_{H,R}(v) = \mathtt{S}(i)$			

$\boxed{M \longmapsto M'}$ (RTCG)

For the rtcg instructions where $(H, R, \iota; I) \longmapsto (H', R'[\mathtt{pc} = \mathtt{S}(i + |\iota|)], I')$

where $R(\texttt{pc}) = \texttt{S}(i)$:

$\iota =$	$H' =$	$R' =$	$I' =$
$\texttt{start}\ [\rho]r$	H	$\{R[r = \texttt{S}(i)], i \mapsto h\}$	$I(\rho \leftarrow i)$
when $i = malloc_{H,R}(2)$, and $i' = i + 1$ and $h = \texttt{tmpl}[\rho' = i']\emptyset.\emptyset$			
$\texttt{end}\ [\rho]r$	$\{H, H_2 + i(\rho' \leftarrow i)\}$	$R''[r = \texttt{S}(i)]$	$I(\rho \leftarrow i)$
when $R(r) = \texttt{S}(l)$ and $R = \{R'', l \mapsto \texttt{tmpl}[\rho' = i]\emptyset.H_2\}$			
$\texttt{copy}\ [\delta]r_1, r_2, v$	H	$R[l_1 = h_1][r_2 = \texttt{S}(k)]$	$I(\delta \leftarrow k)$
when $R(r_1) = \texttt{S}(l_1)$ and $R(l_1) = \texttt{tmpl}[\rho_1 = i_1]\Psi_1.H_1$ and $S_{H,R}(v) = *l_3$ and $H(l_3) = \texttt{tmpl}[\rho_3 = i_3]\Psi'_1.H_3$ and $H' = \{H_1, H_3 + k\}$and $k = \|H_1\|^+$ and $i = realloc_{H,R}(i_1, k, \|H'\|^+)$ and $h_1 = \texttt{tmpl}([\rho_1 = i]\{\Psi_1, \Psi'_1 + k\}.H')(\rho_3 \leftarrow \rho_1 + k)$			
$\texttt{fill}\ r_1, r_2, r_3$	H	$R[l = h]$	I
when $R(r_1) = \texttt{S}(l)$ and $R(r_2) = \texttt{S}(i)$ and $R(r_3) = w$ and $R(l) = \texttt{tmpl}[\rho = i']\{\Psi_1, i \mapsto \tau\}.H'$ and $h = \texttt{tmpl}[\rho = i']\Psi_1.H'[i = w]$			
$\texttt{fillr}\ r_1, r_2, r_3$	H	$R[l_1 = h]$	I
when $R(r_1) = \texttt{S}(l_1)$ and $R(r_2) = \texttt{S}(i_2)$ and $R(r_3) = \texttt{S}(i_3)$ and $R(l_1) = \texttt{tmpl}[\rho = i']\{\Psi_1, i_2 \mapsto \tau\}.H'$ and $h = \texttt{tmpl}[\rho = i']\Psi_1.H'[i_2 = \texttt{rel}[\rho + i_2] * (i_3 - i_2)]$			

$\boxed{M \longmapsto M'}$ (control instructions)

For $(H, R, I) \longmapsto M'$.

$I =$	$M' =$
$\texttt{bnz}\ r, v; I'$	$(H, R[\texttt{pc} = \texttt{S}(l)], I''(\delta_1, \ldots, \delta_n \leftarrow \gamma_1, \ldots, \gamma_n))$
where $S_{H,R}(v) = *l[\gamma_1] \ldots [\gamma_n]$ and $H(l) = \texttt{code}[\delta_1, \ldots, \delta_n]\Gamma.I''$ and $R(r) \neq 0$	
$\texttt{jmp}\ v$	$(H, R[\texttt{pc} = \texttt{S}(l)], I'(\delta_1, \ldots, \delta_n \leftarrow \gamma_1, \ldots, \gamma_n))$
where $S_{H,R}(v) = *l[\gamma_1] \ldots [\gamma_n]$ and $H(l) = \texttt{code}[\delta_1, \ldots, \delta_n]\Gamma''.I'$	
$\texttt{jmpr}\ v$	$(H, R[\texttt{pc} = \texttt{S}(l)], I'(\delta_1, \ldots, \delta_n \leftarrow \gamma_1, \ldots, \gamma_n))$
where $S_{H,R}(v) = (\texttt{rel}[\gamma] * i)[\gamma_1] \ldots [\gamma_n]$ and $R(\texttt{pc}) = \texttt{S}(l')$ and $l = l' + i$ and $H(l) = \texttt{code}[\delta_1, \ldots, \delta_n]\Gamma.I'$	

B.3 Judgements

$\boxed{\vdash (H, R, I)}$

$$\frac{\mathrm{dom}^{+}H \cap \mathrm{dom}^{+}R = \emptyset \qquad \begin{array}{c}\cdot \vdash \Psi \qquad \cdot \vdash \Gamma \\ \cdot; \Psi \vdash H : \Psi\end{array} \qquad \cdot; \Psi \vdash R : \Gamma \qquad \cdot; \Psi; \Gamma \vdash I}{\vdash (H, R, I)}$$

$\boxed{\Delta; \Psi \vdash H : \Psi'}$

$$(\text{H-empty})\frac{}{\Delta; \Psi \vdash \emptyset : \emptyset}$$

$$(\text{H-mem})\frac{\Delta; \Psi \vdash h : \sigma}{\Delta; \Psi \vdash \{l \mapsto h\} : \{l \mapsto \sigma\}}$$

$$(\text{H-reg})\frac{\Delta; \Psi \vdash w : \tau}{\Delta; \Psi \vdash \{r' \mapsto w\} : \{r' \mapsto \tau\}}(r' \in r, \mathtt{pc})$$

$$(\text{H-union})\frac{\Delta; \Psi \vdash H_1 : \Psi_1 \qquad \Delta; \Psi \vdash H_2 : \Psi_2 \qquad \mathrm{dom}^{+}\, H_1 \cap \mathrm{dom}^{+}H_2 = \emptyset}{\Delta; \Psi \vdash \{H_1, H_2\} : \{\Psi_1, \Psi_2\}}$$

$$(\text{H-sub})\frac{\Delta; \Psi \vdash H : \Psi' \qquad \Delta \vdash \Psi' \leq \Psi''}{\Delta; \Psi \vdash H : \Psi''}$$

$\boxed{\Delta; \Psi \vdash h : \sigma}$

$$(\text{h-code})\frac{\Delta, \Delta'; \Psi; \Gamma \vdash I}{\Delta; \Psi \vdash \mathtt{code}[\Delta']\Gamma.I : \forall[\Delta'].\Gamma}(\Delta \cap \Delta' = \emptyset)$$

$$(\text{h-tmpl})\frac{\begin{array}{c}\Delta, \rho; \{\Psi, \Psi_1 + \rho, \Psi_2 + \rho\} \vdash H : \{\Psi_1', \Psi_2\} \qquad \Delta \vdash \{\Psi_1, \Psi_2\} \\ \mathrm{dom}^{+}H = \{i | 0 \leq i \leq k\} \qquad \mathrm{dom}\ \Psi_1 = \mathrm{dom}\Psi_1'\end{array}}{\Delta; \Psi \vdash \mathtt{tmpl}[\rho = l]\Psi_1.H : [\rho](\Psi_1 \Rightarrow \Psi_2)}(\rho \notin \Delta)$$

$$(\text{h-w})\frac{\Delta; \Psi \vdash w : \tau}{\Delta; \Psi \vdash w : \tau} \qquad (\text{h-sub})\frac{\Delta; \Psi \vdash h : \sigma \qquad \Delta \vdash \sigma \leq \sigma'}{\Delta; \Psi \vdash h : \sigma'}$$

$\boxed{\Delta; \Psi \vdash w : \tau}$

$$(\text{w-int})\frac{}{\Delta; \Psi \vdash i : \mathit{int}} \qquad (\text{w-sing})\frac{}{\Delta; \Psi \vdash \mathtt{S}(i) : S(i)}$$

$$(\text{w-ptr})\frac{\Delta \vdash \Psi(\gamma + i) \leq \sigma \quad \sigma@(\gamma + i) = \sigma'}{\Delta; \Psi \vdash \mathtt{rel}[\gamma] * i : \mathit{rel}[\gamma]\ * \sigma'}$$

$$(\text{w-inst})\frac{\Delta;\Psi\vdash w : \mathit{rel}[\gamma]\ *\forall[\delta',\Delta'].\Gamma \qquad \Delta\vdash\gamma'}{\Delta;\Psi\vdash w[\gamma'] : \mathit{rel}[\gamma]\ *\forall[\Delta'].\Gamma(\delta'\leftarrow\gamma')}$$

$$(\text{w-sub})\frac{\Delta;\Psi\vdash w : \tau \qquad \Delta\vdash\tau\leq\tau'}{\Delta;\Psi\vdash w : \tau'}$$

$\boxed{\Delta;\Psi;\Gamma\vdash v : \tau}$

$$(\text{v-reg})\frac{\Delta\vdash\Gamma(r)\leq\tau}{\Delta;\Psi;\Gamma\vdash r : \tau} \qquad (\text{v-w})\frac{\Delta;\Psi\vdash w : \tau}{\Delta;\Psi;\Gamma\vdash w : \tau}$$

$$(\text{v-hole})\frac{\Gamma(\texttt{pc}) = S(\gamma) \qquad \Delta\vdash\Psi(\gamma+1)\leq\tau}{\Delta;\Psi;\Gamma\vdash [] : \tau}$$

$$(\text{v-inst})\frac{\Delta;\Psi;\Gamma\vdash v : \mathit{rel}[\gamma]\ *\forall[\delta',\Delta'].\Gamma \qquad \Delta\vdash\gamma'}{\Delta;\Psi;\Gamma\vdash v[\gamma'] : \mathit{rel}[\gamma]\ *\forall[\Delta'].\Gamma(\delta'\leftarrow\gamma')}$$

$$(\text{v-sub})\frac{\Delta;\Psi\vdash v : \tau \qquad \Delta\vdash\tau\leq\tau'}{\Delta;\Psi\vdash v : \tau'}$$

$\boxed{\sigma@\gamma = \sigma'}$

$$\sigma@\gamma = \begin{cases} \forall[\Delta].\Gamma & \text{if } \sigma = \forall[\Delta].\{\Gamma, \texttt{pc} = S(\gamma)\} \\ \sigma & \text{otherwise.} \end{cases}$$

$\boxed{\Delta;\Psi;\Gamma\vdash I}$

$$(\text{I-inst})\frac{\begin{array}{c}\Gamma(\texttt{pc}) = S(\gamma) \qquad \{\Delta;\Psi;\Gamma\}\ \iota\ \{\Delta';\Psi';\Gamma'\} \\ \Delta';\Psi';\Gamma'[\texttt{pc} = S(\gamma+|\iota|)]\vdash I\end{array}}{\Delta;\Psi;\Gamma\vdash\iota; I}$$

$$(\text{I-jmp})\frac{\Gamma(\texttt{pc}) = S(\gamma) \qquad \Delta;\Psi;\Gamma\vdash v : *\forall[].\Gamma' \qquad \Delta\vdash\Gamma\backslash\texttt{pc}\leq\Gamma'}{\Delta;\Psi;\Gamma\vdash\texttt{jmp}\ v}$$

$$(\text{I-jmpr})\frac{\Gamma(\texttt{pc}) = S(\gamma) \qquad \Delta;\Psi;\Gamma\vdash v : \mathit{rel}[\gamma+1]\ *(\forall[].\Gamma') \qquad \Delta\vdash\Gamma\backslash\texttt{pc}\leq\Gamma'}{\Delta;\Psi;\Gamma\vdash\texttt{jmpr}\ v}$$

$\boxed{\{\Delta;\Psi;\Gamma\}\ \iota\ \{\Delta';\Psi';\Gamma'\}}$ (no RTCG)

$$\frac{\Gamma(r_2) = \mathit{int} \qquad \Delta;\Psi\vdash v : \mathit{int}}{\{\Delta;\Psi;\Gamma\}\ \ \texttt{add}\ r_1, r_2, v\ \{\Delta;\Psi;\Gamma[r_1 = \mathit{int}]\}}$$

$$\frac{\Gamma(r_2) = S(\gamma_1) \qquad \Delta;\Psi;\Gamma\vdash v : S(\gamma_2)}{\{\Delta;\Psi;\Gamma\}\ \ \texttt{add}\ r_1, r_2, v\ \{\Delta;\Psi;\Gamma[r_1 = S(\gamma_1+\gamma_2)]\}}$$

$$\frac{\Gamma(r) = int \qquad \Delta;\Psi \vdash v : *\forall[].\Gamma' \qquad \Delta \vdash \Gamma\backslash\texttt{pc} \leq \Gamma'}{\{\Delta;\Psi;\Gamma\}\ \texttt{bnz}\ r, v\ \{\Delta;\Psi;\Gamma\}}$$

$$\frac{\Delta;\Psi \vdash v : \tau}{\{\Delta;\Psi;\Gamma\}\ \texttt{mov}\ r, v\ \{\Delta;\Psi;\Gamma[r = \tau]\}}$$

$$\frac{\begin{array}{c}\Delta;\Psi;\Gamma \vdash v : S(\gamma)\\ \Psi(\gamma) = \sigma \qquad \sigma@\gamma = \sigma'\end{array}}{\{\Delta;\Psi;\Gamma\}\ \texttt{forget}\ r, v\ \{\Delta;\Psi;\Gamma[r = *\sigma']\}}$$

$\boxed{\{\Delta;\Psi;\Gamma\}\ \iota\ \{\Delta';\Psi';\Gamma'\}}$ (RTCG)

$$\frac{\Gamma' = \{\Gamma, \rho \mapsto [\rho'].\emptyset \Rightarrow \emptyset\}}{\{\Delta;\Psi;\Gamma\}\ \texttt{start}\ [\rho]r\ \{\Delta,\rho;\Psi;\Gamma'[r = S(\rho)]\}}(\rho \notin \Delta)$$

$$\frac{\Gamma(r) = S(\gamma) \qquad \Gamma = \{\Gamma', \gamma \mapsto [\rho'].(\emptyset \Rightarrow \Psi')\}}{\{\Delta;\Psi;\Gamma\}\ \texttt{end}\ [\rho]r\ \{\Delta,\rho;\{\Psi, \Psi' + \rho(\rho' \leftarrow \rho)\};\Gamma'[r = S(\rho)]\}}(\rho \notin \Delta)$$

$$\frac{\begin{array}{c}\Gamma(r_1) = S(\gamma) \quad \Gamma = \{\Gamma', \gamma \mapsto [\rho''].(\Psi_1 \Rightarrow \Psi_2)\} \quad \Gamma(v) = *[\rho'].(\Psi_1' \Rightarrow \Psi_2')\\ \sigma = [\rho''].(\{\Psi_1, \Psi_1' + \delta(\rho' \leftarrow \rho'' + \delta)\} \Rightarrow \{\Psi_2, \Psi_2' + \delta(\rho' \leftarrow \rho'' + \delta)\})\end{array}}{\{\Delta;\Psi;\Gamma\}\ \texttt{copy}\ [\delta]r_1, r_2, v\ \{\Delta,\delta;\Psi;\{\Gamma', \gamma \mapsto \sigma\}[r_2 = S(\delta)]\}}(\delta \notin \Delta)$$

$$\frac{\begin{array}{c}\Gamma(r_1) = S(\gamma_1) \qquad \Gamma(r_2) = S(\gamma_2) \qquad \Delta;\Psi;\Gamma \vdash v : \tau\\ \Gamma = \{\Gamma', \gamma_1 \mapsto [\rho'].\{\Psi', \gamma_2 \mapsto \tau\} \Rightarrow \Psi''\}\end{array}}{\{\Delta;\Psi;\Gamma\}\ \texttt{fill}\ r_1, r_2, v\ \{\Delta;\Psi;\{\Gamma', \gamma_1 \mapsto [\rho'].\Psi' \Rightarrow \Psi''\}\}}(\rho' \notin \Delta)$$

$$\frac{\begin{array}{c}\Gamma(r_1) = S(\gamma_1) \qquad \Gamma(r_2) = S(\gamma_2) \qquad \Gamma(r_3) = S(\gamma_3)\\ \Psi''(\gamma_3) = \sigma \qquad \sigma@\rho' + \gamma_3 = \sigma'\\ \Gamma = \{\Gamma', \gamma_1 \mapsto [\rho'].(\{\Psi', \gamma_2 \mapsto rel[\rho' + \gamma_2]\ *\sigma'\} \Rightarrow \Psi'')\end{array}}{\{\Delta;\Psi;\Gamma\}\ \texttt{fillr}\ r_1, r_2, r_3\ \{\Delta;\Psi;\{\Gamma', \gamma_1 \mapsto [\rho'].(\Psi' \Rightarrow \Psi'')\}\}}(\rho' \notin \Delta)$$

$\boxed{\Delta \vdash \Psi_1 \leq \Psi_2,\ \Delta \vdash \sigma \leq \sigma'}$

$$\frac{}{\Delta \vdash \sigma \leq \sigma} \qquad \frac{}{\Delta \vdash \tau \leq junk}$$

$$\frac{\Delta \vdash \Psi_2 \leq \Psi_2'}{\Delta \vdash [\rho]\Psi_1 \Rightarrow \Psi_2 \leq [\rho]\Psi_1 \Rightarrow \Psi_2'}$$

$$\frac{\Delta \vdash \sigma_1 \leq \sigma_1' \quad \ldots \quad \Delta \vdash \sigma_k \leq \sigma_k'}{\Delta \vdash \{\eta_1 \mapsto \sigma_1, \ldots, \eta_n \mapsto \sigma_n\} \leq \{\eta_1 \mapsto \sigma_1', \ldots, \eta_k \mapsto \sigma_k'\}}(0 \leq k \leq n)$$

$\boxed{\Delta \vdash \Psi,\ \Delta \vdash \sigma}$

$$\frac{\Delta \vdash \eta_1 \quad \ldots \quad \Delta \vdash \eta_n \qquad \Delta \vdash \sigma_1 \quad \ldots \quad \Delta \vdash \sigma_n}{\Delta \vdash \{\eta_1 \mapsto \sigma_1, \ldots, \eta_n \mapsto \sigma_n\}} (\eta_i \text{ distinct})$$

$$\frac{FTV(\gamma) \subseteq \Delta}{\Delta \vdash \gamma} \qquad \frac{FTV(\sigma) \subseteq \Delta}{\Delta \vdash \sigma}$$

$\boxed{\mathrm{dom}^+ H,\ |H|^+}$

$$\begin{array}{rcl} \mathrm{dom}^+\{H_1, H_2\} & = & \mathrm{dom}^+ H_1 \cup \mathrm{dom}^+ H_2 \\ \mathrm{dom}^+\{l \mapsto \mathtt{code}[\Delta]\Gamma.I\} & = & W(l, I) \\ \mathrm{dom}^+\{l' \mapsto \mathtt{tmpl}[\rho = l]\Psi_1.H\} & = & \{l'\} \cup \{l + i | i \in \mathrm{dom}^+ H\} \end{array}$$

where

$$\begin{array}{rcl} W(l, \iota; I) & = & \{l\} \cup W(l + |\iota|, I) \\ W(l, \mathtt{jmp}\ r) & = & \{l\} \end{array}$$

$|H|^+ = k$ whenever $\mathrm{dom}^+ H = \{i | 0 \le i < k\}$. $\boxed{S_{H,R}(v)}$

$$S_{H,R}(v) = \begin{cases} R(r) & \text{when } v = r \\ w & \text{when } v = w \\ S_{H,R}(v')[\gamma] & \text{when } v = v'[\gamma] \\ w & \text{when } v = [] \text{ and } R(\mathtt{pc}) = \mathtt{S}(i) \\ & \text{ and } H(i+1) = w \end{cases}$$

$\boxed{|\iota|}$

$$|\iota| = \begin{cases} 1 + |v| & \text{when } v \text{ occurs in } \iota \\ 1 & \text{otherwise} \end{cases} \quad \text{and } |v| = \begin{cases} 1 & \text{when } v = [] \\ |v'| & \text{when } v = v'[\gamma] \\ 0 & \text{otherwise.} \end{cases}$$

$\boxed{H + i,\ \Psi + \gamma}$

$$\begin{array}{rcl} \emptyset + i & = & \emptyset \\ \{l \mapsto h\} + i & = & \{l + i \mapsto h'\} \\ & & \text{where } h' = h \text{ unless } h = \mathtt{tmpl}[\rho = l] \ldots \\ & & \text{in which case } h' = \mathtt{tmpl}[\rho = l + i] \ldots \\ \{H_1, H_2\} + i & = & \{H_1 + i, H_2 + i\} \end{array}$$

$$\begin{array}{rcl} \emptyset + \gamma & = & \emptyset \\ \{\gamma' \mapsto \sigma\} + \gamma & = & \{\gamma' + \gamma \mapsto \sigma\} \\ \{H_1, H_2\} + \gamma & = & \{H_1 + \gamma, H_2 + \gamma\} \end{array}$$

Appendix C

TAL/T: Soundness Proof

This chapter sketches out the proof of soundness for TAL/T. The Soundness Theorem states that TAL/T's operational semantics never become stuck when executing well-typed programs. This fact implies that programs never access memory they have not allocated, and never attempt to execute arbitrary data as code.

We will prove Soundness using a Preservation Lemma, which states that well-typed programs step to well-typed programs, and a Progress Lemma, which states that well-typed programs can always take a step.

Lemma 2 (Preservation) *If* $\vdash M$ *and* $M \longmapsto M'$ *then* $\vdash M'$.

Proof: In Section C.4. $\square$

Lemma 3 (Progress) *If* $\vdash M$ *then there is some* M' *such that* $M \longmapsto M'$.

Proof: In Section C.5. $\square$

This chapter develops the machinery needed to prove Preservation and Progress. Given these two lemmas, Soundness is easily established.

Theorem 4 (Soundness) *If* $\vdash M$ *and* $M \longmapsto \ldots \longmapsto M'$ *then there is some* M'' *such that* $M' \longmapsto M''$.

Proof: By induction on the number of reduction steps, using the Preservation Lemma, $\vdash M'$. By the Progress Lemma, M' is not stuck. $\square$

The proofs of Preservation and Progress are largely syntactic. Most of the arguments for these and supporting lemmas are simple inductions on derivations or exhaustive case analysis over syntactic forms. The nature of these proofs makes them largely uninteresting. In the following sections, we present the harder cases, explain the intended proof procedure, and gloss over the uninteresting cases. All the real work goes into formulating the lemmas not into proving them.

Please refer to Chapter 9 for a verbal description of TAL/T. Appendix B collects all the pieces of the formal TAL/T definition: the syntax, operational rules, and type-checking rules. We often refer to rules by name. The names are given next to the definitions of the rules.

We group the required lemmas into sections by function: standard lemmas, heap lemmas, and template lemmas. The proofs of Preservation and Progress are given each in their own section.

C.1 Standard Lemmas

These standard lemmas occur in almost every proof of soundness using this proof technique. The Substitution Lemma ensures that after instantiation, the body of a polymorphic function remains well-typed and, in our case, that templates remain well-typed when copied.

Lemma 5 (Substitution) *If* $\Delta \vdash \gamma$ *then*

1. *If* $\delta, \Delta; \Psi \vdash H : \Psi'$ *then* $(\Delta; \Psi \vdash H : \Psi')(\delta \leftarrow \gamma)$.
2. *If* $\delta, \Delta; \Psi \vdash h : \sigma$ *then* $(\Delta; \Psi \vdash h : \sigma)(\delta \leftarrow \gamma)$.
3. *If* $\delta, \Delta; \Psi \vdash w : \tau$ *then* $(\Delta; \Psi \vdash w : \tau)(\delta \leftarrow \gamma)$.
4. *If* $\delta, \Delta; \Psi; \Gamma \vdash v : \tau$ *then* $(\Delta; \Psi; \Gamma \vdash v : \tau)(\delta \leftarrow \gamma)$.
5. *If* $\delta, \Delta; \Psi; \Gamma \vdash I$ *then* $(\Delta; \Psi; \Gamma \vdash I)(\delta \leftarrow \gamma)$
6. *If* $\{\delta, \Delta; \Psi; \Gamma\}\ \iota\ \{\delta, \Delta'; \Psi'; \Gamma'\}$ *then* $(\{\Delta; \Psi; \Gamma\}\ \iota\ \{\Delta'; \Psi'; \Gamma'\})(\delta \leftarrow \gamma)$
7. *If* $\delta, \Delta \vdash \sigma \leq \sigma'$ *then* $(\Delta \vdash \sigma \leq \sigma')(\delta \leftarrow \gamma)$.
8. *If* $\delta, \Delta \vdash \Psi_1 \leq \Psi_2$ *then* $(\Delta \vdash \Psi_1 \leq \Psi_2)(\delta \leftarrow \gamma)$.
9. *If* $\delta, \Delta \vdash \gamma'$ *then* $\Delta \vdash \gamma'(\delta \leftarrow \gamma)$.
10. *If* $\sigma @ \gamma' = \sigma'$ *then* $(\sigma @ \gamma' = \sigma')(\delta \leftarrow \gamma)$.

Proof:

The proof proceeds by simultaneous structural induction on each of the derivations. A new derivation tree is constructed by propagating the substitution to the leaves of the tree. The cases are for the most part immediate.

One subtlety has to do with multi-sets. Several rules have assumptions of the form $\Psi(\gamma') = \sigma$. After substitution we need to prove that $(\Psi(\delta \leftarrow \gamma))(\gamma'(\delta \leftarrow \gamma)) = \sigma(\delta \leftarrow \gamma)$. This fact follows from properties of multi-sets. $\Psi(\gamma') = \sigma$ implies that $\gamma' \mapsto \sigma \in \Psi$. By the definition of substitution, $\gamma'(\delta \leftarrow \gamma) \mapsto \sigma(\delta \leftarrow \gamma) \in \Psi(\delta \leftarrow \gamma)$. Because lookup $\Psi(\gamma)$ is defined non-deterministically, we can choose to extract this particular element rather than any other that may apply after substitution.

The rule (h-tmpl) is another tricky case. That rule has as a condition that $\Delta \vdash \{\Psi_1, \Psi_2\}$. But well-formedness is not in general preserved by substitution. However, in this case $\text{dom}\{\Psi_1, \Psi_2\} \subseteq \text{dom}H$ which by the Well-formed Heaps Lemma (Lemma 15), implies that $FTV(\{\Psi_1, \Psi_2\}) = \emptyset$. For this special case, substitution cannot change the domain of the multi-set and therefore well-formedness is preserved.

□

The Value-Type Lemma connects values and their types. It is important when proving the Preservation Lemma because it allows us to determine the typing judgement that was used from the value that triggered the operational rule.

Lemma 6 (Value-Type) *If* $\cdot; \Psi \vdash h : \sigma$ *or* $\cdot; \Psi \vdash w : \tau$ *then*

1. *If* $w = i$ *then* $\cdot \vdash int \leq \tau$.
2. *If* $w = \mathtt{S}(i)$ *then* $\cdot \vdash S(i) \leq \tau$.
3. *If* $w = \mathtt{rel}[i] * l$ *then* $\cdot \vdash rel[i] * \sigma' \leq \tau$.
4. *If* $h = \mathtt{tmpl}[\rho = i]\Psi_1.H$ *then* $\sigma = [\rho]\Psi_1 \Rightarrow \Psi_2$.
5. *If* $h = \mathtt{code}[\delta_1, \ldots, \delta_n]\Gamma.I$ *then* $\sigma = \forall[\delta_1, \ldots, \delta_n].\Gamma$ *and* $\mathtt{pc} \in \text{dom}\Gamma$.

Proof:

The proof is by induction on the typing derivation. In each case the only rules type-checking a value with the shape of w or h is either the subtyping rule or a specific rule immediately satisfying the right-hand side. In the cases where the implication indicates equality rather than subtyping, all possible subtypes have the same shape.

The special condition for code types is used to ensure that the $\mathtt{pc}$ register is set properly on jumps. It is easy to prove. Every rule for blocks requires $\mathtt{pc} \in \text{dom}\Gamma$.

□

The Canonical Forms Lemma is the converse of the Value-Type Lemma. It allows us to predict from the type the shape of the value. This capability is used when proving Progress to ensure that the conditions needed to trigger an operational rule are satisfied.

Lemma 7 (Canonical Forms) *If* $\cdot \vdash \Psi$, $\cdot; \Psi \vdash h : \sigma$ *and* $\cdot; \Psi \vdash w : \tau$ *then*

1. *if* $\tau = int$ *then* $w = i$
2. *if* $\tau = S(l)$ *then* $w = \mathtt{S}(l)$
3. *if* $\tau = rel[k] * \sigma'$ *then* $w = \mathtt{rel}[k] * l$ *unless* $\sigma' = \forall[\Delta].\Gamma$ *in which case* $w = \mathtt{rel}[k] * l[\gamma_1] \ldots [\gamma_n]$.

4. *if* $\sigma = \forall[\Delta].\Gamma$ *then* $h = \texttt{code}[\Delta]\Gamma.I$

5. *if* $\sigma = [\rho].\Gamma \Rightarrow \Psi'$ *then* $h = \texttt{tmpl}[\rho = l]\Gamma.H$.

Proof:

The proof proceeds by independent structural induction on the typing derivation in each case. We begin by considering all rules that might produce types of the relevant shape. In each case the rules (w-sub) and/or (h-sub) apply.

From the rules for subtyping it follows that if $\Delta \vdash \tau \leq \tau'$ and $\tau' \neq junk$ then $\tau = \tau'$. Similarly, if $\Delta \vdash \sigma \leq \sigma'$ where σ' is not τ then either $\sigma' = \sigma$ or σ and σ' are both template types. These facts allow the induction hypothesis to be applied for (w-sub) and (h-sub).

The remaining cases are then straightforward.

□

The Well-formedness Lemma simply states that typing judgements only produce well-formed types.

Lemma 8 (Well-formedness) *For any* Δ, Ψ *and* Γ *such that* $\Delta \vdash \Psi$ *and* $\Delta \vdash \Gamma$ *all the following hold:*

1. *If* $\Delta; \Psi \vdash w : \tau$ *then* $\Delta \vdash \tau$.

2. *If* $\Delta; \Psi; \Gamma \vdash v : \tau$ *then* $\Delta \vdash \tau$.

3. *If* $\Delta; \Psi \vdash h : \sigma$ *then* $\Delta \vdash \sigma$.

Proof: Follows directly by induction on the derivations. □

Most subtyping relationships are explicitly transitive. Ours is not so we must prove transitivity. Transitivity is useful for callapsing proof trees that make multiple uses of subsumption.

Lemma 9 (Transitive Subtyping) 1. *If* $\Delta \vdash \sigma_1 \leq \sigma_2$ *and* $\Delta \vdash \sigma_2 \leq \sigma_3$ *then* $\Delta \vdash \sigma_1 \leq \sigma_3$.

2. *If* $\Delta \vdash \Psi_1 \leq \Psi_2$ *and* $\Delta \vdash \Psi_2 \leq \Psi_3$ *then* $\Delta \vdash \Psi_1 \leq \Psi_3$

Proof: Immediate. □

Lemma 10 (Subtyping Invariance) *If* $\Delta \vdash \Psi'$ *and* $\Delta \vdash \Psi' \leq \Psi$ *then*

1. $\Delta \vdash \Psi$

2. *If* $\Psi(\eta) = \sigma$ *then* $\Psi'(\eta) = \sigma'$ *and* $\Delta \vdash \sigma' \leq \sigma$

3. $\Delta \vdash \Psi'[\eta = \sigma] \leq \Psi[\eta = \sigma]$

Proof: All three properties are immediate consequences of the subtyping rule for heap types. □

Lemma 11 (Context Strengthening) *If*

$$\Delta \subseteq \Delta' \qquad \Delta \vdash \Psi' \leq \Psi \qquad \Delta \vdash \Gamma' \leq \Gamma$$

then

1. *If* $\Delta; \Psi \vdash H : \Psi''$ *then* $\Delta'; \Psi' \vdash H : \Psi''$.
2. *If* $\Delta; \Psi \vdash h : \sigma$ *then* $\Delta'; \Psi' \vdash h : \sigma$.
3. *If* $\Delta; \Psi \vdash w : \tau$ *then* $\Delta'; \Psi' \vdash w : \tau$.
4. *If* $\Delta; \Psi; \Gamma \vdash I$ *then* $\Delta'; \Psi'; \Gamma' \vdash I$.
5. *If* $\{\Delta; \Psi; \Gamma\}\ \iota\ \{\Delta''; \Psi_a; \Gamma_a\}$ *then* $\{\Delta'; \Psi'; \Gamma'\}\ \iota\ \{\Delta'' \cup \Delta'; \Psi'_a; \Gamma'_a\}$ *and* $\Delta'' \cup \Delta' \vdash \Psi'_a \leq \Psi_a$ *and* $\Delta'' \cup \Delta' \vdash \Gamma'_a \leq \Gamma_a$
6. *If* $\Delta \vdash \sigma \leq \sigma'$ *then* $\Delta' \vdash \sigma \leq \sigma'$
7. *If* $\Delta \vdash \Psi_1 \leq \Psi_2$ *then* $\Delta' \vdash \Psi_1 \leq \Psi_2$
8. *If* $\Delta \vdash \sigma$ *then* $\Delta' \vdash \sigma$
9. *If* $\Delta \vdash \Psi''$ *then* $\Delta' \vdash \Psi''$

Proof:

The proof proceeds by simultaneous induction on the structure of the derivation for each of the above. Most of the cases are uninteresting.

In all cases where a judgement requires that a location map to a particular type, the only possible subtype is the type itself so the rule is unaffected by replacing Ψ with Ψ'. When the type is unconstrained, subtyping is always available to coerce the type in Ψ' if necessry.

The extra subtyping conditions are needed in the case of instructions ι so that the induction for blocks works. Note that Δ' and Δ'' are ordinary sets not multi-sets as are all elements of the syntactic class Δ.

□

C.2 Heap Lemmas

These heap related lemmas prove properties about heaps that are essential for ensuring that the actual heap and the type given to that heap remain synchronized under a variety of updates. Many of these lemmas rely on strong assumptions about well-formedness to ensure that the heap types in question are partial functions.

Lemma 12 (Heap-Type) *If*

$$\cdot \vdash \Psi, \quad \cdot; \Psi \vdash H : \Gamma, \quad H(L) = h, \quad \Gamma(L) = \sigma$$

then $\cdot; \Psi \vdash h : \sigma$.

Proof:

The proof proceeds by induction on the structure of the derivation $\cdot; \Psi \vdash H : \Gamma$. For the base cases, rules (H-mem) and (H-reg), the result is immediate. For (H-union), L occurs in either one branch or the other, but not both. Therefore, the result follows by induction. For the rule (H-sub), our induction hypothesis gives us that $\cdot; \Psi \vdash h : \sigma'$ where $\cdot \vdash \Gamma' \leq \Gamma$ and $\Gamma'(L) = \sigma'$. By the subtyping rule for heaps, $\cdot \vdash \sigma' \leq \sigma$ so we can apply the rule (h-sub) to conclude that $\cdot; \Psi \vdash h : \sigma$.

□

Lemma 13 (Canonical Heaps) *If* $\Delta; \Psi \vdash H : \{\Psi_1, \Psi_2\}$ *then there are* H_1 *and* H_2 *such that* $H = \{H_1, H_2\}$ *and* $\Delta; \Psi \vdash H_1 : \Psi_1$ *and* $\Delta; \Psi \vdash H_2 : \Psi_2$

Proof:

The proof proceeds by induction on the typing derivation for H. The cases for (H-empty), (H-mem) and, (H-reg) are immediate since in those cases either Ψ_1, Ψ_2, or both must be empty.

In the case of (H-union), we have by assumption that $H = \{H'_1, H'_2\}$ and $\Psi = \{\Psi'_1, \Psi'_2\}$. Furthermore $\Delta; \Psi \vdash H'_1 : \Psi'_1$, $\Delta; \Psi \vdash H'_2 : \Psi'_2$, and $\mathrm{dom}^+ H'_1 \cap \mathrm{dom}^+ H'_2 = \emptyset$.

Because Ψ also equals $\{\Psi_1, \Psi_2\}$, we can construct (using standard properties of multi-sets) $\Psi_{11}, \Psi_{12}, \Psi_{21}, \Psi_{22}$ such that all the following hold:

$$\begin{array}{rclrcl} \Psi_1 & = & \{\Psi_{11}, \Psi_{12}\} & \Psi'_1 & = & \{\Psi_{11}, \Psi_{21}\} \\ \Psi_2 & = & \{\Psi_{21}, \Psi_{22}\} & \Psi'_2 & = & \{\Psi_{12}, \Psi_{22}\} \end{array}$$

Applying the induction hypothesis, we construct H_{ij} such that $\Delta; \Psi \vdash H_{ij} : \Psi_{ij}$.

We aim to prove that $\Delta; \Psi \vdash \{H_{11}, H_{12}\} : \Psi_1$ and that $\Delta; \Psi \vdash \{H_{21}, H_{22}\} : \Psi_2$. From our induction hypothesis and the above equations, we have almost all we need to apply (H-union) except that $\mathrm{dom}^+ H_{i1} \cap \mathrm{dom}^+ H_{i2} = \emptyset$. But this follows from definition of dom^+ and the following facts: $H_{ij} \subseteq H'_j$, and $\mathrm{dom}^+ H'_1 \cap \mathrm{dom}^+ H'_2 = \emptyset$.

The remaining case is that for (H-sub). Using Lemma 14 the result is almost immediate. The induction hypothesis gives us H_1 and H_2 as required but they have types Ψ'_1 and Ψ'_2 such that $\Delta \vdash \Psi'_1 \leq \Psi_1$ and $\Delta \vdash \Psi'_2 \leq \Psi_2$. Applying (H-sub) yields the required result.

$\square$

Lemma 14 (Heap Subtyping) *If $\Delta \vdash \Psi \leq \{\Psi_1, \Psi_2\}$ then there are Ψ'_1 and Ψ'_2 such that $\Psi = \{\Psi'_1, \Psi'_2\}$ and $\Delta \vdash \Psi'_1 \leq \Psi_1$ and $\Delta \vdash \Psi'_2 \leq \Psi_2$.*

Proof: There is only one subtyping rule for heaps. The result is immediate. $\square$

Lemma 15 (Well-formed Heaps) *If $\Delta \vdash \Psi$ and $\Delta; \Psi \vdash H : \Psi'$ then*

1. $\mathrm{dom}\Psi' \subseteq \mathrm{dom}H \subseteq \mathrm{dom}^+H$.
2. *H is a partial function.*
3. *Ψ' is a partial function.*
4. $\Delta \vdash \Psi'$

Proof:

The fact that $\mathrm{dom}H \subseteq \mathrm{dom}^+H$ is an immediate consequence of the definitions of dom and dom^+. That H is a partial function is easily shown by induction. The only rule that combines two heaps is (H-union) but that rule requires the two heaps to be disjoint. That Ψ' is a partial function is a consequence of $\mathrm{dom}\Psi' \subseteq \mathrm{dom}^+H$ and the same inductive argument as for H. Finally Ψ' is well-formed because Ψ' is a partial function and the leaves of the derivation (by the Well-formedness Lemma (Lemma 8)) only produce well-formed types.

The only unproven fact is that $\mathrm{dom}\Psi' \subseteq \mathrm{dom}^+H$. Induction on the typing derivation for Ψ' completes the proof.

$\square$

Lemma 16 (Heap Lookup) *If*

$$\cdot \vdash \Psi, \quad \cdot \vdash \Gamma, \quad \cdot; \Psi \vdash H : \Psi, \quad \cdot; \Psi \vdash R : \Gamma, \quad \mathrm{dom}^+H \cap \mathrm{dom}^+R = \emptyset$$

then

1. $\Gamma(L) = \tau$ *implies* $R(L) = w$ *and* $\cdot; \Psi \vdash w : \tau$.
2. $\Psi(L) = \sigma$ *implies* $H(L) = h$ *and* $\cdot; \Psi \vdash h : \sigma$.
3. $\cdot; \Psi; \Gamma \vdash v : \tau$ *implies* $\cdot; \Psi \vdash S_{H,R}(v) : \tau$.

Proof:

The first two cases are proven identically. By the Well-formed Heaps Lemma (Lemma 15), Γ, Ψ, H, and R are all partial functions and $\mathrm{dom}\Gamma \subseteq \mathrm{dom}R$ and $\mathrm{dom}\Psi \subseteq \mathrm{dom}H$. Therefore $\Gamma(L), R(L), \Psi(L)$, and $H(L)$ are all uniquely defined.

By the Heap-Type Lemma (Lemma 12), $\cdot; \Psi \vdash w : \tau$ and $\cdot; \Psi \vdash h : \sigma$.

The final case for v is somewhat tricky. The cases for $v = r$, $v = w$ follow immediately from the previous cases. The case $v = v'[\chi]$ follows by induction on the depth of instantiations.

The only interesting case is therefore $v = []$. For $[]$ to type-check, $\Gamma(\texttt{pc}) = \mathit{ptr}(\gamma)$ for some γ. Since there are no type variables in scope, γ must be an integer i. By the previous cases $R(\texttt{pc}) = w$ such that $\cdot; \Psi \vdash w : \mathit{ptr}(i)$. By the Canonical Forms Lemma, $w = \texttt{ptr}(i)$. For $[]$ to type-check we must also have that $\cdot \vdash \Psi(i+1) \leq \tau$. By the previous case, $H(i+1) = h$ such that $\cdot; \Psi \vdash h : \Psi(i+1)$. One application of (v-sub) yields the desired result.

□

Lemma 17 (Heap Extension) *If*

$$\Delta \vdash \Psi, \quad \Delta; \Psi \vdash H : \Psi, \quad \Delta; \{\Psi, \Psi''\} \vdash H' : \{\Psi', \Psi''\}, \quad \mathrm{dom}^{+}H \cap \mathrm{dom}^{+}H' = \emptyset$$

then $\Delta; \{\Psi, \Psi', \Psi''\} \vdash \{H, H'\} : \{\Psi, \Psi', \Psi''\}$.

Proof:

Let Ψ_f denote $\{\Psi, \Psi', \Psi''\}$. By Lemma 15 it follows that $\mathrm{dom}\Psi \subseteq \mathrm{dom}^{+}H$ and $\mathrm{dom}\{\Psi', \Psi''\} \subseteq \mathrm{dom}^{+}H'$ and therefore that $\mathrm{dom}\Psi \cap \mathrm{dom}\{\Psi', \Psi''\} = \emptyset$. This fact implies that $\Delta \vdash \Psi_f \leq \Psi$ and that $\Delta \vdash \Psi_f \leq \{\Psi, \Psi''\}$ so by the Context Strengthening Lemma (Lemma 11) we have that

$$\Delta; \Psi_f \vdash H : \Psi \text{ and } \Delta; \Psi_f \vdash H' : \{\Psi', \Psi''\}$$

Then by the rule (H-union), we get the desired result that $\Delta; \Psi_f \vdash \{H, H'\} : \Psi_f$.

□

Lemma 18 (Heap Retraction) *If* $\Delta; \Psi \vdash H : \Psi'$ *and* $H = \{H_1, H_2\}$ *then there exist* Ψ'_1 *and* Ψ'_2 *such that* $\Psi' = \{\Psi'_1, \Psi'_2\}$ *and* $\Delta; \Psi \vdash H_1 : \Psi'_1$ *and* $\Delta; \Psi \vdash H_2 : \Psi'_2$.

Proof:

The proof proceeds by induction on the derivation of $\Delta; \Psi \vdash H : \Psi'$. In this argument we rely on the Well-formed Heaps Lemma (Lemma 15) to conclude that Ψ', Ψ'_1, and Ψ'_2 are partial functions. The base case for the rules (H-reg) and

(H-mem) are immediate as H_1 or H_2 must be $\emptyset$, and therefore one of Ψ'_1 or Ψ'_2 is Ψ' and the other is $\emptyset$.

Consider the case for the rule (H-union). This rule splits the heap H into two parts H'_1 and H'_2 having types Ψ''_1 and Ψ''_2 respectively such that $\Psi' = \{\Psi''_1, \Psi''_2\}$.

Let $H'_{ij} = H'_i \cap H_j$ for $i = 1, 2$ and $j = 1, 2$. Then $H'_1 = \{H'_{11}, H'_{12}\}$ and $H'_2 = \{H'_{21}, H'_{22}\}$, so we can apply our induction hypothesis to conclude that there are decompositions $\Psi''_1 = \{\Psi''_{11}, \Psi''_{12}\}$ and $\Psi''_2 = \{\Psi''_{21}, \Psi''_{22}\}$.

Note that by the definition of $\mathrm{dom}^+ H$, $H_{ij} \subseteq H'_i$ implies that $\mathrm{dom}^+ H_{ij} \subseteq \mathrm{dom}^+ H'_i$. Since $\mathrm{dom}^+ H'_1 \cap \mathrm{dom}^+ H'_2 = \emptyset$, $\mathrm{dom}^+ H_{1j} \cap \mathrm{dom}^+ H_{2j} = \emptyset$. Therefore we can apply the rule (H-union) to conclude that $\Delta; \Psi \vdash \{H'_{11}, H'_{21}\} : \{\Psi''_{11}, \Psi''_{21}\}$ and that $\Delta; \Psi \vdash \{H'_{12}, H'_{22}\} : \{\Psi''_{12}, \Psi''_{22}\}$. But by definition, $H_1 = \{H_{11}.H_{21}\}$ and $H_2 = \{H'_{12}, H'_{22}\}$, thus we already have the required decomposition in hand.

The remaining case for the rule (H-sub) follows immediately.

□

Lemma 19 (Heap Replacement) *If* $\Delta; \Psi \vdash R : \Gamma$ *and* $\Delta; \Psi \vdash w : \tau$ *then* $\Delta; \Psi \vdash R[L = w] : \Gamma[L = \tau]$

Proof: By structural induction on the derivation $\Delta; \Psi \vdash R : \Gamma$. □

Lemma 20 (Heap Invariance) *If* $\Delta; \Psi \vdash H : \Psi'$ *then* $\Delta; \Psi \vdash H + i : \Psi' + i$ *for any* i.

Proof: Immediate. □

Lemma 21 (Code Placement) *If* $\cdot \vdash \Psi$ *and* $\cdot \vdash \gamma_i$ *for* $1 \leq i \leq n$ *and* $\cdot; \Psi \vdash \mathtt{rel}[i] * l[\gamma_1] \ldots [\gamma_n] : \mathit{rel}[i] * \forall[\Delta'].\Gamma'$ *then* $\Psi(i+l)@(i+l) = \forall[\delta_n, \ldots, \delta_1, \Delta'].\Gamma''$ *such that* $\Gamma' = \Gamma''[\delta_n \ldots \delta_1 \leftarrow \gamma_1, \ldots, \gamma_n]$.

Proof:

By induction on the typing derivation and the number of instantiations. The only rules that apply are (w-ptr), (w-inst) or (w-sub). The case for rule (w-sub) is immediate as the only subtype of a type with this shape is itself. The case for the rule (w-inst) follows from the induction hypothesis. The base case is the rule (w-ptr). Since there are no subtypes of code types the result follows.

□

C.3 Template Lemmas

The template lemmas capture the most important properties of templates. The Template Typing Lemma shows that typing derivations for template values can always be collapsed to one use of the rule (h-sub) and one use of (h-tmpl). The Template Position Invariance Lemma establishes the key property of templates – they can be moved. Finally the Template Copy and the Template Fill Lemmas capture the key properties required by their respective operations.

Lemma 22 (Template Typing) *If*

$$\Delta; \Psi \vdash \texttt{tmpl}[\rho = l]\Psi_1.H : [\rho]\Psi_1 \Rightarrow \Psi_2$$

then there is a derivation of the form

$$\textit{(h-tmpl)}\frac{\cdots}{\Delta; \Psi \vdash \texttt{tmpl}[\rho = l]\Psi_1.H : [\rho]\Psi_1 \Rightarrow \Psi_2'}$$

such that $\Delta \vdash \Psi_2' \leq \Psi_2$

Proof:

By induction on the derivation. Only the rules (h-tmpl) and (h-sub) can produce a derivation for a value with the shape `tmpl`.... If the final rule in the derivation uses (h-tmpl) the proof is complete. Otherwise the final rule must be (h-sub).

The only sub-type of a type with the form $[\rho]\Psi_1 \Rightarrow \Psi_2$ is a type $[\rho]\Psi_1 \Rightarrow \Psi_2'$ such that $\Delta \vdash \Psi_2' \leq \Psi_2$. By the Transitive Subtyping Lemma (Lemma 9) and our induction hypothesis, the result follows.

□

Lemma 23 (Template Position Invariance) *If*

$$\Delta; \Psi \vdash \texttt{tmpl}[\rho = l]\Psi_1.H : [\rho]\Psi_1 \Rightarrow \Psi_2 \textit{ and } \Delta \vdash \gamma + k$$

then there is a Ψ_2' *such that* $\Delta \vdash \Psi_2' \leq \Psi_2$ *and*

$$(\Delta; \{\Psi, \Psi_1 + \gamma + k, \Psi_2' + \gamma + k\} \vdash H + k : \{\Psi_1' + k, \Psi_2' + k\})(\rho \leftarrow \gamma + k)$$

and $\mathrm{dom}\Psi_1' = \mathrm{dom}\Psi_1$ *and* $\Delta \vdash \{\Psi_1, \Psi_2'\}(\rho \leftarrow \gamma + k)$.

Proof:

By Lemma 22, we pick a derivation ending with the rule (h-tmpl) and having type $[\rho]\Psi_1 \Rightarrow \Psi_2'$ such that $\Delta \vdash \Psi_2' \leq \Psi_2$. By the rule (h-tmpl)

$\Delta, \rho; \{\Psi, \Psi_1 {+} \rho, \Psi_2' {+} \rho\} \vdash H : \{\Psi_1', \Psi_2'\}$ and $\mathrm{dom}\Psi_1 = \mathrm{dom}\Psi_1'$ and $\rho, \Delta \vdash \{\Psi_1, \Psi_2'\}$

Then we apply the Heap Invariance Lemma (Lemma 20) to conclude that

$$\Delta, \rho; \{\Psi, \Psi_1 + \rho, \Psi_2' + \rho\} \vdash H + k : \{\Psi_1' + k, \Psi_2' + k\}$$

The final result follows by a direct application of the Substitution Lemma (Lemma 5).

$$(\Delta; \{\Psi, \Psi_1 + \gamma + k, \Psi_2' + \gamma + k\} \vdash H + k : \{\Psi_1' + k, \Psi_2' + k\})(\rho \leftarrow \gamma + k)$$

□

Lemma 24 (Template Copy) *If all the following hold.*

1. $\Delta;\Psi \vdash \mathtt{tmpl}[\rho = i].\Psi_1.H : [\rho]\Psi_1 \Rightarrow \Psi_2$
2. $\Delta;\Psi \vdash \mathtt{tmpl}[\rho' = i'].\Psi'_1 \Rightarrow \Psi'_2.H' : [\rho']\Psi'_1 \Rightarrow \Psi'_2$
3. $k = |H|^+$

Then (abbreviate $\Psi''_1 = \{\Psi_1, \Psi'_1{+}k\}(\rho' \leftarrow \rho{+}k)$ *and* $\Psi''_2 = \{\Psi_2, \Psi'_2{+}k\}(\rho' \leftarrow \rho{+}k)$*)*

$$\Delta;\Psi \vdash \mathtt{tmpl}[\rho = i''].\Psi''_1.\{H, H' + k\}(\rho' \leftarrow \rho + k) : [\rho].\Psi''_1 \Rightarrow \Psi''_2$$

Proof:

The first step in the proof is to use the Template Typing Lemma (Lemma 22) so that we only have to consider derivations ending in the rule (h-tmpl). By the Template Typing Lemma, there are heap types Ψ_3 and Ψ'_3 such that $\Delta \vdash \Psi_3 \leq \Psi_2$ and $\Delta \vdash \Psi'_3 \leq \Psi'_2$; and derivations ending in (h-tmpl) giving the templates the types $[\rho]\Psi_1 \Rightarrow \Psi_3$ and $[\rho']\Psi'_1 \Rightarrow \Psi'_3$ respectively.

We now complete the proof using these types and the assumption that the final step in the derivation is an application of (h-tmpl). From this assumption, we can conclude

$$\begin{array}{ll} \Delta,\rho;\{\Psi,\Psi_1+\rho,\Psi_3+\rho\} \vdash H : \{\Psi_4,\Psi_3\} & \Delta \vdash \{\Psi_1,\Psi_3\} \\ \Delta,\rho';\{\Psi,\Psi'_1+\rho',\Psi'_3+\rho'\} \vdash H' : \{\Psi'_4,\Psi'_3\} & \Delta \vdash \{\Psi'_1,\Psi'_3\} \end{array}$$

Applying the Context Strengthening Lemma, the Substitution Lemma, and the Position Invariance Lemma to the second fact above we obtain

$$(\Delta,\rho;\{\Psi,\Psi'_1+\rho+k,\Psi'_3+\rho+k\} \vdash H'+k : \{\Psi'_4+k,\Psi'_3+k\})(\rho' \leftarrow \rho+k)$$

Let $\Psi'_5 = \{\Psi'_1 + k, \Psi'_3 + k\}(\rho' \leftarrow \rho + k)$, and $\Psi_5 = \{\Psi_1, \Psi_3\}$. By the definition of k, $\mathrm{dom}\Psi'_5 \cap \mathrm{dom}\Psi_5 = \emptyset$. Furthermore each of these heap types is well-formed, so the combined heap

$$\Psi''_6 = \{\Psi_5, \Psi'_5\}$$

is well-formed and furthermore $\Delta,\rho \vdash \Psi''_6 \leq \Psi'_5$ and $\Delta,\rho \vdash \Psi''_6 \leq \Psi_5$. By the Context Strengthening Lemma, we can conclude that

$$\begin{array}{l} \Delta,\rho;\Psi''_6+\rho \vdash H : \{\Psi_4,\Psi_3\} \\ (\Delta,\rho;\Psi''_6+\rho \vdash H'+k : \{\Psi'_4+k,\Psi'_3+k\})(\rho' \leftarrow \rho+k) \end{array}$$

FIX ME: Implicit use of a context-weakening lemma to eliminate ρ'.

By construction $\text{dom}^+ H \cap \text{dom}^+(H' + k) = \emptyset$, therefore we can apply (H-union) to conclude that

$$(\Delta, \rho; \Psi_6'' + \rho \vdash \{H, H' + k\} : \{\Psi_4, \Psi_4', \Psi_3, \Psi_3'\})(\rho' \leftarrow \rho + k)$$

With all definitions unrolled, this derivation is the key condition needed to prove

$$\Delta; \Psi \vdash \texttt{tmpl}[\rho = i]\Psi_1''.\{H, H' + k\}(\rho' \leftarrow \rho + k) : [\rho]\Psi_1'' \Rightarrow \Psi_3''$$

where $\Psi_1'' = \{\Psi_1, \Psi_1'\}(\rho' \leftarrow \rho + k)$ and $\Psi_3'' = \{\Psi_3, \Psi_3''\}(\rho' \leftarrow \rho + k)$. The other conditions required by rule (h-tmpl) are trivial to verify.

We have proved the lemma for Ψ_3'' but need to prove it for Ψ_2'' (as defined in the statement of the lemma). But by construction $\Delta \vdash \Psi_3'' \leq \Psi_2''$. Therefore we can use the rule (h-sub) to obtain the desired result.

□

Lemma 25 (Template Fill) *If all the following*

1. $\Delta; \Psi \vdash \texttt{tmpl}[\rho = i].\Psi_1.H : [\rho]\Psi_1 \Rightarrow \Psi_2$
2. $\Delta, \rho; \{\Psi, \Psi_1 + \rho, \Psi_2 + \rho\} \vdash w : \tau$
3. $\Psi_1 = \{\Psi_1', k \mapsto \tau\}$

then $\Delta; \Psi \vdash \texttt{tmpl}[\rho = i].\Psi_1'.H[k \mapsto w] : [\rho]\Psi_1' \Rightarrow \Psi_2$

Proof:

By the Template Typing Lemma (Lemma 22), there is a derivation ending in the (h-tmpl) rule that $\Delta; \Psi \vdash \texttt{tmpl} \ldots : [\rho]\Psi_1 \Rightarrow \Psi_2'$ and $\Delta \vdash \Psi_2' \leq \Psi_2$. Using the Context Strengthening Lemma (Lemma 11), we can establish the other properties for Ψ_2'. Therefore it suffices to prove the lemma for the case where $\Psi_2 = \Psi_2'$ and the derivation of $\Delta; \Psi \vdash \texttt{tmpl} \ldots : [\rho]\Psi_1 \Rightarrow \Psi_2$ ends in the rule (h-tmpl).

By the rule (h-tmpl),

$$\Delta, \rho; \{\Psi, \Psi_1 + \rho, \Psi_2 + \rho\} \vdash H : \{\Psi_1'', \Psi_2\} \quad \Delta, \rho \vdash \{\Psi_1, \Psi_2\} \quad \text{dom}\ \Psi_1 = \text{dom}\Psi_1''$$

By the Heap Replacement Lemma (Lemma 19),

$$\Delta, \rho; \{\Psi, \Psi_1 + \rho, \Psi_2 + \rho\} \vdash H[k = w] : \{\Psi_1''[k = \tau], \Psi_2\}$$

But this term can be rewritten as

$$\Delta, \rho; \{\Psi, \Psi_1' + \rho, \{\Psi_2, k \mapsto \tau\} + \rho\} \vdash H[k = w] : \{\Psi_1'' \backslash k, \{\Psi_2, k \mapsto \tau\}\}$$

Therefore, using the rule (h-tmpl) we can conclude,

$$\Delta; \Psi \vdash \texttt{tmpl}[\rho = i].\Psi_1'.H[k \mapsto w] : [\rho]\Psi_1' \Rightarrow \{\Psi_2, k \mapsto \tau\}$$

The desired result follows by one application of the (h-sub) rule.

□

C.4 Proof of Preservation

Recall the statement of the theorem. If $\vdash M$ and $M \longmapsto M'$ then $\vdash M'$. In the following let $M = (H, R, \iota; I)$. Technically for control-flow instructions there is no ι: But for the sake of uniformity we beg the reader to forgive this slight trespass.

From the assumption $\vdash (H, R, \iota; I)$ we conclude there exist Ψ and Γ such that:

1. $\cdot \vdash \Psi$ and $\cdot \vdash \Gamma$
2. $\cdot; \Psi \vdash H : \Psi$
3. $\cdot; \Psi \vdash R : \Gamma$
4. $\mathrm{dom}^{+} H \cap \mathrm{dom}^{+} R = \emptyset$
5. $\cdot; \Psi; \Gamma \vdash \iota; I$

To prove the lemma we will use these facts in conjunction with the lemmas proven in earlier sections to conclude that $\vdash M'$ is derivable after the operational step. We prove the theorem by case analysis – one case for each operational rule. By assumption the operational step can be taken, and therefore all its requirements are met. In each case we repeat the relevant operational rule (OR), and typing judgement (TJ). Although we show the complete typing judgement, our assumptions imply that $\Delta = \emptyset$ in every case.

For every non-control-flow instructions there is a proof obligation to show that the $\mathtt{pc}$ register may be updated by the size of the instruction. Rather than proving this fact in each case we dispense with it for once here.

More formally, for this set of instructions $M' = (H', R'[\mathtt{pc} = \mathtt{S}(i + |\iota|)], I)$ and, because $\iota; I$ is well-typed, the typing judgement below also holds.

$$\text{(I-inst)}\frac{\Gamma(\mathtt{pc}) = S(\gamma) \qquad \{\Delta; \Psi; \Gamma\}\ \iota\ \{\Delta'; \Psi'; \Gamma'\} \qquad \Delta'; \Psi'; \Gamma'[\mathtt{pc} = S(\gamma + |\iota|)] \vdash I}{\Delta; \Psi; \Gamma \vdash \iota; I}$$

Recall that $\Delta = \emptyset$. By the Well-formed Heaps Lemma (Lemma 15) and the definition of well-formedness, γ must be a plain integer, call it i. By the Heap Lookup Lemma (Lemma 16) and the Canonical Forms Lemma (Lemma 7), $R(\mathtt{pc}) = \mathtt{S}(i)$. By the Heap Replacement Lemma (Lemma 19) (and the Substitution Lemma (Lemma 5) when $\Delta' \neq \Delta$), assuming we have proven the step otherwise correct, $\cdot; \Psi' \vdash R'[\mathtt{pc} = \mathtt{S}(i + |\iota|)] : \Gamma'[\mathtt{pc} = S(i + |\iota|)]$.

In the remaining cases we will assume that types are always well-formed without comment. This fact is always easy to prove. We will also implicitly assume that $\cdot; \Psi \vdash R(L) : \Gamma(L)$ and that $\cdot; \Psi \vdash H(L) : \Psi(L)$ as this fact follows directly from the Heap Lookup Lemma.

case $\texttt{add}(1)$:

OR:

$$\frac{\iota = \quad H' = \quad R' = \quad I' =}{\texttt{add}\ r_1, r_2, v \quad H \quad R[r_1 = R(r_2) + S_{H,R}(v)] \quad I}$$

TJ:

$$\frac{\Gamma(r_2) = \mathit{int} \qquad \Delta; \Psi \vdash v : \mathit{int}}{\{\Delta; \Psi; \Gamma\}\ \ \texttt{add}\ r_1, r_2, v\ \{\Delta; \Psi; \Gamma[r_1 = \mathit{int}]\}}$$

By the Value-Type Lemma, $\Gamma(r_2) = \mathit{int}$. By the Heap-Replacement Lemma, $\cdot; \Psi \vdash R[r_1 = i] : \Gamma[r_1 = \mathit{int}]$ holds for any integer result of the addition.

case $\texttt{add}(2)$:

OR:

$$\frac{\iota = \quad H' = \quad R' = \quad I' =}{\texttt{add}\ r_1, r_2, v \quad H \quad R[r_1 = \texttt{S}(i + j)] \quad I}$$
$$\text{when } R(r_2) = \texttt{S}(i) \text{ and } S_{H,R}(v) = \texttt{S}(j)$$

TJ:

$$\frac{\Gamma(r_2) = S(\gamma_1) \qquad \Delta; \Psi; \Gamma \vdash v : S(\gamma_2)}{\{\Delta; \Psi; \Gamma\}\ \ \texttt{add}\ r_1, r_2, v\ \{\Delta; \Psi; \Gamma[r_1 = S(\gamma_1 + \gamma_2)]\}}$$

By the Value-Type Lemma, $\Gamma(r_2) = S(i)$. Just as in the previous case, the Heap-Replacement Lemma allows us to conclude that P' is well-typed.

case $\texttt{bnz}$: (not taken)

OR:

$$\frac{\iota = \quad H' = \quad R' = \quad I' =}{\texttt{bnz}\ r, v \quad H \quad R \quad I}$$
$$\text{when } R(r) = 0$$

TJ:

$$\frac{\Gamma(r) = \mathit{int} \qquad \Delta; \Psi \vdash v : *\forall[].\Gamma' \qquad \Delta \vdash \Gamma \backslash \texttt{pc} \leq \Gamma'}{\{\Delta; \Psi; \Gamma\}\ \ \texttt{bnz}\ r, v\ \{\Delta; \Psi; \Gamma\}}$$

Trivial.

case $\texttt{bnz}$: (taken)

OR:

$$\frac{\iota = \quad M' =}{\texttt{bnz}\ r, v; I' \quad (H, R[\texttt{pc} = \texttt{S}(l)], I''(\delta_1, \ldots, \delta_n \leftarrow \gamma_1, \ldots, \gamma_n))}$$
$$\text{where } S_{H,R}(v) = *l[\gamma_1] \ldots [\gamma_n]$$
$$\text{and } H(l) = \texttt{code}[\delta_1, \ldots, \delta_n]\Gamma.I'' \text{ and } R(r) \neq 0$$

TJ:

$$\frac{\Gamma(r) = \mathit{int} \qquad \Delta; \Psi \vdash v : *\forall[].\Gamma' \qquad \Delta \vdash \Gamma \backslash \texttt{pc} \leq \Gamma'}{\{\Delta; \Psi; \Gamma\}\ \ \texttt{bnz}\ r, v\ \{\Delta; \Psi; \Gamma\}}$$

This case is similar to that for `jmp`.

case `mov`:

OR:

$$\frac{\iota = \quad H' = \quad R' = \quad I' =}{\texttt{mov}\ r, v \quad H \quad R[r = S_{H,R}(v)] \quad I}$$

TJ:

$$\frac{\Delta; \Psi \vdash v : \tau}{\{\Delta; \Psi; \Gamma\}\ \texttt{mov}\ r, v\ \{\Delta; \Psi; \Gamma[r = \tau]\}}$$

By the Heap Lookup Lemma $\cdot; \Psi \vdash S_{H,R}(v) : \tau$, and by the Heap-Replacement Lemma the desired result follows immediately.

case `forget`:

OR:

$$\frac{\iota = \quad H' = \quad R' = \quad I' =}{\texttt{forget}\ r, v \quad H \quad R[r = *i] \quad I}$$
$$\text{when } S_{H,R}(v) = \texttt{S}(i)$$

TJ:

$$\frac{\Delta; \Psi; \Gamma \vdash v : S(\gamma) \qquad \Psi(\gamma) = \sigma \qquad \sigma@\gamma = \sigma'}{\{\Delta; \Psi; \Gamma\}\ \texttt{forget}\ r, v\ \{\Delta; \Psi; \Gamma[r = *\sigma']\}}$$

By the Value-Type Lemma the type of $S_{H,R}(v)$ is $S(i)$. From TJ, we know that $\Psi(i) = \sigma$ and that $\sigma@i = \sigma'$. It therefore follows that $\cdot; \Psi \vdash *i : *\sigma'$.

The result follows directly by the Heap Replacement Lemma.

case `start`:

OR:

$$\frac{\iota = \quad H' = \quad R' = \quad I' =}{\texttt{start}\ [\rho]r \quad H \quad \{R[r = \texttt{S}(i)], i \mapsto h\} \quad I(\rho \leftarrow i)}$$
$$\text{when } i = malloc_{H,R}(2),\ \text{and } i' = i + 1$$
$$\text{and } h = \texttt{tmpl}[\rho' = i']\emptyset.\emptyset$$

TJ:

$$\frac{\Gamma' = \{\Gamma, \rho \mapsto [\rho'].\emptyset \Rightarrow \emptyset\}}{\{\Delta; \Psi; \Gamma\}\ \texttt{start}\ [\rho]r\ \{\Delta, \rho; \Psi; \Gamma'[r = S(\rho)]\}} (\rho \notin \Delta)$$

We wish to add the heap $R'' = \{i \mapsto \texttt{tmpl}[\rho' = i']\emptyset.\emptyset\}$ to R, with type $\Gamma'' = \{i \mapsto [\rho']\emptyset \Rightarrow \emptyset\}$. It is easy to verify that

$$\cdot; \Psi \vdash R'' : \Gamma''$$

By the definition of *malloc*, the extended domain of this heap does not intersect the domains of R or H. By the rule (H-union) and the Heap Replacement Lemma, $\cdot; \Psi \vdash \{R[r = \texttt{S}(i)], R''\} : \{\Gamma[r = S(i)], \Gamma''\}$. After updating the `pc` and applying the substitution lemma we get the desired result.

case `end`:

OR:

$$\frac{\iota = \quad\quad H' = \quad\quad R' = \quad\quad I' =}{\texttt{end}\,[\rho]r \quad \{H, H_2 + i(\rho' \leftarrow i)\} \quad R''[r = \texttt{S}(i)] \quad I(\rho \leftarrow i)}$$
$$\text{when } R(r) = \texttt{S}(l) \text{ and } R = \{R'', l \mapsto \texttt{tmpl}[\rho' = i]\emptyset.H_2\}$$

TJ:

$$\frac{\Gamma(r) = S(\gamma) \qquad \Gamma = \{\Gamma', \gamma \mapsto [\rho'].(\emptyset \Rightarrow \Psi')\}}{\{\Delta; \Psi; \Gamma\} \;\; \texttt{end}\,[\rho]r \; \{\Delta, \rho; \{\Psi, \Psi' + \rho(\rho' \leftarrow \rho)\}; \Gamma'[r = S(\rho)]\}} (\rho \notin \Delta)$$

By the Heap Retraction Lemma (Lemma 18), there is a Γ'' such that $\cdot; \Psi \vdash R'' : \Gamma''$. By the Value-Type Lemma,

$$\cdot; \Psi \vdash \texttt{tmpl}[\rho' = i]\emptyset.H_2 : [\rho']\emptyset \Rightarrow \Psi_2$$

We must check a number of conditions to conclude $\vdash (H', R'[\texttt{pc} = \ldots], I)$.

First we apply the Template Position Invariance Lemma (Lemma 23) to conclude that

$$(\cdot; \{\Psi, \Psi_2' + i\} \vdash H_2 + i : \{\Psi_2' + i\})(\rho' \leftarrow i)$$

for some Ψ_2' such that $\cdot \vdash \Psi_2' \leq \Psi_2$.

Let $\Psi' = \{\Psi, \Psi_2' + i(\rho' \leftarrow i)\}$, and $\Gamma' = \Gamma''(\rho' \leftarrow i)$ in the following analysis.

1. $\mathrm{dom}^+ H' \cap \mathrm{dom}^+ R' = \emptyset$

   By definition of dom^+, $\mathrm{dom}^+ R' = \mathrm{dom}^+ R'' = \mathrm{dom}^+ R \backslash \{\mathrm{dom}^+(H_2 + i), l\}$, and $\mathrm{dom}^+ H' = \mathrm{dom}^+ H \cup \mathrm{dom}^+(H_2 + i(\rho' \leftarrow i))$. By assumption $\mathrm{dom}^+ H \cap \mathrm{dom}^+ R = \emptyset$ from which it follows that $\mathrm{dom}^+ H' \cap \mathrm{dom}^+ R' = \emptyset$.

2. $\cdot; \Psi' \vdash H' : \Psi'$

   Applying the Heap Extension Lemma (Lemma 17) would yield the result $\cdot; \Psi' \vdash H' : \Psi'$. However to apply this lemma we must prove that $\mathrm{dom}^+ H \cap \mathrm{dom}^+(H_2 + i)(\rho' \leftarrow i) = \emptyset$. But that follows directly from the fact that $\mathrm{dom}^+(H_2 + i) \subseteq \mathrm{dom}^+ R$ and $\mathrm{dom}^+ H \cap \mathrm{dom}^+ R = \emptyset$.

3. $\cdot \vdash \Psi'$

   Since $FTV(\Psi') \subseteq \emptyset$ and $\cdot; \Psi' \vdash H' : \Psi'$, it follows from Lemma 15 that $\cdot \vdash \Psi'$.

4. $\cdot; \Psi' \vdash R' : \Gamma'$

   Since $\cdot \vdash \Psi'$ and Ψ' extends Ψ it follows that $\cdot \vdash \Psi' \leq \Psi$ and therefore by the Context Strengthening Lemma we obtain $\cdot; \Psi' \vdash R'' : \Gamma''$. By the Heap Replacement Lemma $\cdot; \Psi' \vdash R''[r = \texttt{S}(i)] : \Gamma''[r = S(i)]$.

5. $\cdot;\Psi';\Gamma'[\mathtt{pc}=\ldots]\vdash I(\rho'\leftarrow i)$
   The continuation I is type-checked in a context with Ψ_2 but we have Ψ_2'. Fortunately, $\cdot\vdash\Psi_2'\leq\Psi_2$ so we can apply the Context Strengthening Lemma followed by the Substitution Lemma to get the required result.

case `copy`:
OR:

$\iota =$	$H' =$	$R' =$	$I' =$
$\mathtt{copy}\ [\delta]r_1,r_2,v$	H	$R[l_1=h_1][r_2=\mathtt{S}(k)]$	$I(\delta\leftarrow k)$

when $R(r_1)=\mathtt{S}(l_1)$ and $R(l_1)=\mathtt{tmpl}[\rho_1=i_1]\Psi_1.H_1$
and $S_{H,R}(v)=*l_3$ and $H(l_3)=\mathtt{tmpl}[\rho_3=i_3]\Psi_1'.H_3$
and $H'=\{H_1,H_3+k\}$and $k=|H_1|^+$ and $i=\mathit{realloc}_{H,R}(i_1,k,|H'|^+)$
and $h_1=\mathtt{tmpl}([\rho_1=i]\{\Psi_1,\Psi_1'+k\}.H')(\rho_3\leftarrow\rho_1+k)$

TJ:

$$\frac{\begin{array}{c}\Gamma(r_1)=S(\gamma)\quad \Gamma=\{\Gamma',\gamma\mapsto[\rho''].(\Psi_1\Rightarrow\Psi_2)\}\quad \Gamma(v)=*[\rho'].(\Psi_1'\Rightarrow\Psi_2')\\ \sigma=[\rho''].(\{\Psi_1,\Psi_1'+\delta(\rho'\leftarrow\rho''+\delta)\}\Rightarrow\{\Psi_2,\Psi_2'+\delta(\rho'\leftarrow\rho''+\delta)\})\end{array}}{\{\Delta;\Psi;\Gamma\}\ \ \mathtt{copy}\ [\delta]r_1,r_2,v\ \{\Delta,\delta;\Psi;\{\Gamma',\gamma\mapsto\sigma\}[r_2=S(\delta)]\}}(\delta\notin\Delta)$$

We use the Value-Type Lemma to conclude that the templates $R(l_1)$ and $H(l_3)$ are well-typed in the heap. By the Template Copy Lemma the resulting value h_1 is well-typed as well. Abbreviate the type of h_1 by σ_1.

Let $\Gamma'=\Gamma[l_1=\sigma_1][r_2=S(k)]$. Then the Heap Replacement Lemma, allows us to conclude that $\cdot;\Psi\vdash R':\Gamma'$. The heap type Γ' is exactly the result of substituting k for δ in the final Γ of TJ. Therefore by the Substitution Lemma, $I(\delta\leftarrow k)$ will type-check under Γ'.

It remains to be shown that the domain of R' does not intersect the domain of H. The definition of *realloc* ensures that this fact holds.

case `fill`:
OR:

$\iota =$	$H' =$	$R' =$	$I' =$
$\mathtt{fill}\ r_1,r_2,r_3$	H	$R[l=h]$	I

when $R(r_1)=\mathtt{S}(l)$ and $R(r_2)=\mathtt{S}(i)$ and $R(r_3)=w$
and $R(l)=\mathtt{tmpl}[\rho=i']\{\Psi_1,i\mapsto\tau\}.H'$
and $h=\mathtt{tmpl}[\rho=i']\Psi_1.H'[i=w]$

TJ:

$$\frac{\begin{array}{c}\Gamma(r_1)=S(\gamma_1)\qquad \Gamma(r_2)=S(\gamma_2)\qquad \Delta;\Psi;\Gamma\vdash v:\tau\\ \Gamma=\{\Gamma',\gamma_1\mapsto[\rho'].\{\Psi',\gamma_2\mapsto\tau\}\Rightarrow\Psi''\}\end{array}}{\{\Delta;\Psi;\Gamma\}\ \ \mathtt{fill}\ r_1,r_2,v\ \{\Delta;\Psi;\{\Gamma',\gamma_1\mapsto[\rho'].\Psi'\Rightarrow\Psi''\}\}}(\rho'\notin\Delta)$$

By the Value-Type Lemma, we have that $\cdot;\Psi \vdash \texttt{tmpl}[\rho = i']\{\Psi_1, i_2 \mapsto \tau\}.H' : [\rho]\{\Psi_1, i_2 \mapsto \tau\}{\Rightarrow}\Psi_2$ and that $\cdot;\Psi \vdash w : \tau'$. By the type-checking rule for fill, $\tau = \tau'$. Applying Context Strengthening Lemma we can satisfy the conditions for the Template Fill Lemma to conclude that $\cdot;\Psi \vdash h : [\rho]\Psi_1{\Rightarrow}\Psi_2$.

The result follows directly similar to previous cases.

case `fillr`:
OR:

$$\frac{\iota = \qquad\qquad H' = \quad R' = \qquad I' =}{\texttt{fillr}\, r_1, r_2, r_3 \quad H \quad R[l_1 = h] \quad I}$$
$$\text{when } R(r_1) = \texttt{S}(l_1) \text{ and } R(r_2) = \texttt{S}(i_2) \text{ and } R(r_3) = \texttt{S}(i_3)$$
$$\text{and } R(l_1) = \texttt{tmpl}[\rho = i']\{\Psi_1, i_2 \mapsto \tau\}.H'$$
$$\text{and } h = \texttt{tmpl}[\rho = i']\Psi_1.H'[i_2 = \texttt{rel}[\rho + i_2] * (i_3 - i_2)]$$

TJ:

$$\frac{\begin{array}{c}\Gamma(r_1) = S(\gamma_1) \qquad \Gamma(r_2) = S(\gamma_2) \qquad \Gamma(r_3) = S(\gamma_3) \\ \Psi''(\gamma_3) = \sigma \qquad \sigma@\rho' + \gamma_3 = \sigma' \\ \Gamma = \{\Gamma', \gamma_1 \mapsto [\rho'].(\{\Psi', \gamma_2 \mapsto \mathit{rel}[\rho' + \gamma_2] * \sigma'\}{\Rightarrow}\Psi'')\end{array}}{\{\Delta;\Psi;\Gamma\}\ \texttt{fillr}\, r_1, r_2, r_3\ \{\Delta;\Psi;\{\Gamma', \gamma_1 \mapsto [\rho'].(\Psi'{\Rightarrow}\Psi'')\}\}}(\rho' \notin \Delta)$$

This case is very similar to the case for `fill` except the Template Fill Lemma does not apply directly. By the Value-Type Lemma we know that the template is well-typed as are the values in the various registers. Because the instruction sequence is well-typed, inspection of the rule for `fillr` implies that $\Psi_1(i_2) = \mathit{rel}[\rho + i_2] * \sigma$, and $\Psi_2(i_3) = \sigma'$ such that $\sigma'@\rho + i_2 = \sigma$.

To apply the Template Fill Lemma we must show that that $\rho;\Psi,\Psi_1 + \rho,\Psi_2 + \rho \vdash @[\rho + i_2](i_3 - i_2) : \mathit{rel}[\rho + i_2] * \sigma$. But this fact follows immediately from the rule (w-ptr).

case `jmp`:
OR:

$$\frac{\iota = \qquad M' =}{\texttt{jmp}\ v \quad (H, R[\texttt{pc} = \texttt{S}(l)], I'(\delta_1, \ldots, \delta_n \leftarrow \gamma_1, \ldots, \gamma_n))}$$
$$\text{where } S_{H,R}(v) = *l[\gamma_1]\ldots[\gamma_n] \text{ and } H(l) = \texttt{code}[\delta_1, \ldots, \delta_n]\Gamma''.I'$$

TJ:

$$(\text{I-jmp})\frac{\Gamma(\texttt{pc}) = S(\gamma) \qquad \Delta;\Psi;\Gamma \vdash v : *\forall[].\Gamma' \qquad \Delta \vdash \Gamma\backslash\texttt{pc} \leq \Gamma'}{\Delta;\Psi;\Gamma \vdash \texttt{jmp}\ v}$$

By the Heap Lookup Lemma, $\cdot;\Psi \vdash *l[\gamma_1]\ldots[\gamma_n] : *\forall[].\Gamma'$. Furthermore by TJ, $\cdot \vdash \Gamma\backslash\texttt{pc} \leq \Gamma'$ and therefore $\texttt{pc} \notin \mathrm{dom}\Gamma'$.

Also by the Heap Lookup Lemma, $\texttt{code}[\delta_1, \ldots, \delta_n]\Gamma''.I'$ is well-typed and by the Value-Type Lemma $\texttt{pc} \in \mathrm{dom}\Gamma''$.

By the Code Placement Lemma (Lemma 21), $l@\forall[\delta_1, \ldots, \delta_n]\Gamma'' = \forall[\delta_1, \ldots, \delta_n]\Gamma'''$ for some Γ''' such that $\Gamma' = \Gamma'''(\delta_1, \ldots, \delta_n \leftarrow \gamma_1, \ldots, \gamma_n)$. By the definition of $l@\sigma$ and because $\texttt{pc} \in \mathrm{dom}\Gamma''$ it follows that $\Gamma'' = \{\Gamma''', \texttt{pc} \mapsto S(l)\}$.

From the Substitution Lemma we can conclude that $\cdot; \Psi \vdash R[\texttt{pc} = \texttt{S}(l)] : \{\Gamma', \texttt{pc} \mapsto S(l)\}$. Because the function is well-typed, $\cdot; \Psi; \{\Gamma', \texttt{pc} \mapsto S(l)\} \vdash I'(\delta_1, \ldots, \delta_n \leftarrow \gamma_1, \ldots, \gamma_n)$ also holds.

case $\texttt{jmpr}$:

OR:

$$\frac{\iota = \qquad M' =}{\texttt{jmpr}\ v \quad (H, R[\texttt{pc} = \texttt{S}(l)], I'(\delta_1, \ldots, \delta_n \leftarrow \gamma_1, \ldots, \gamma_n))}$$

where $S_{H,R}(v) = (\texttt{rel}[\gamma] * i)[\gamma_1] \ldots [\gamma_n]$ and $R(\texttt{pc}) = \texttt{S}(l')$ and $l = l' + i$ and $H(l) = \texttt{code}[\delta_1, \ldots, \delta_n]\Gamma.I'$

TJ:

$$(\text{I-jmpr})\frac{\Gamma(\texttt{pc}) = S(\gamma) \qquad \Delta; \Psi; \Gamma \vdash v : rel[\gamma + 1] * (\forall[].\Gamma') \qquad \Delta \vdash \Gamma \backslash \texttt{pc} \leq \Gamma'}{\Delta; \Psi; \Gamma \vdash \texttt{jmpr}\ v}$$

This case is similar to the case for $\texttt{jmp}$.

C.5 Proof of Progress

From the assumption $\vdash (H, R, I)$ we can conclude that there exist Ψ and Γ such that:

1. $\cdot \vdash \Psi$ and $\cdot \vdash \Gamma$
2. $\mathrm{dom}^+ H \cap \mathrm{dom}^+ R = \emptyset$
3. $\cdot; \Psi \vdash H : \Psi$
4. $\cdot; \Psi \vdash R : \Gamma$
5. $\cdot; \Psi; \Gamma \vdash I$

The proof proceeds by induction on the structure of I. Each case relies on the Canonical Forms lemma (Lemma 7 abbreviated CF) and the Heap Lookup lemma(Lemma 16 abbreviated HL). We implicitly apply the Well-formedness Lemma (Lemma 8) to conclude that none of the types contain free type variables.

We use the leading word in I to specify the case.

case `add` (1): By fact 5, $\Gamma(r_2) = int$ and $\cdot; \Psi \vdash v : int$. By CF and HL, $R(r_2) = i$ and $S_{H,R}(v) = j$.

case `add` (2): Similar to `add` (1).

case `bnz`: Similar to the case for `jmp`.

case `mov`: Similar to `add`(1).

case `forget`: Similar to `add` (1).

case `start`: This operational rule always applies.

case `end`: By fact 5, we have that $\Gamma(r) = S(l)$ and $\Gamma = \{\Gamma', l \mapsto [\rho'].(\emptyset \Rightarrow \Psi')$. By the HL and CF, we conclude that $R(l) = \mathtt{tmpl}[\rho'].\emptyset.H$. So the step can be taken.

case `copy`: This case is similar to `end`.

case `fill`: Similar to `end`.

case `fillr`: Similar to `end`.

case `jmp`: By fact 5 and HL, $\cdot; \Psi \vdash S_{H,R}(v) : *\forall[].\Gamma'$. By CF, $S_{H,R}(v) = *l[\gamma_1] \ldots [\gamma_n]$. By the Code Placement Lemma, $\Psi(l) = \forall[\delta_1, \ldots, \delta_n]\Gamma''.I'$. Therefore by HL and CF, $H(l) = \mathtt{code}[\delta_1, \ldots, \delta_n]\Gamma''.I'$ and so the step can be taken.

case `jmpr`: Similar to the case for `jmp`.