Automatic Program Specialization for Interactive Media

Scott Draves

July 23, 1997

CMU-CS-97-159

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Thesis Committee:

Peter Lee, Chair
William Scherlis
Andy Witkin
Olivier Danvy

This research was sponsored in part by the Defense Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software," ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Keywords: Compiler generation, Partial evaluation, Run-time code generation, Bit-addressing, Signal processing

Abstract

This dissertation introduces and analyzes techniques for writing programs that manipulate interactive media. By “media” I mean audio, images, and video flowing through a personal computer. By “interactive” I mean that there is an impatient and unpredictable user who produces and consumes this media. Historically, such systems that provide low latency and remain highly flexible have been rare and difficult to build. I propose treating media systems as programming languages and bringing the techniques of semantics-based program transformation to bear. One part of this solution is the use of compiler generation as an interface to run-time code generation.

The idea is to use *automatic specialization* to convert flexible programs into fast programs. This idea has already been developed by the partial evaluation community. In order to make it work better with media, I extend the known techniques for specialization with partially static integers and equality constraints. I demonstrate the effectiveness of these techniques by using a prototype implementation to benchmark kernels such as wave-table audio synthesis and color-to-grayscale conversion.

Contents

Acknowledgements

Many people helped me write this dissertation. I want to thank the Department of Defense and the Fox Project for funding me. I picked Peter Lee as my advisor because he works on programming languages without losing touch with reality. I made the right choice for the wrong reason: more than anything else, Peter's dedication to teaching and his faith in The System pulled me through. The quality of my writing reflects his high standards and willingness to read many, many drafts. The rest of my committee consisted of Andy Witkin, Bill Scherlis, and Olivier Danvy. Andy advised me when I first arrived at CMU, and continued to support me with access to the animation lab and all its fine equipment. I remember my first meeting with Bill because he was the first faculty I met who also believed the promise of program transformation. Later, we concentrated on writing strategy and he helped me find my voice. Olivier proved to be an excellent outside member. First, he invited me to Denmark where I wrote my first paper. Second, he actually read my work and gave me detailed feedback while tolerating my lack of formality. Besides my committee, I received plenty of technical assistance and education from the CMU-SCS community at large. I am especially grateful to the Zephyr community for their humor and helping me typeset this document. Finally, I want to thank my friends, loves, and family.

There is another, much larger community that also deserves acknowledgement: the free software community. This document and the research it describes were produced with software that is freely available with source code to everyone. In particular, I used linux, emacs, scheme48, gcc, sml/nj, scsh, tex/latex, dot, gnuplot, jgraph, xfig, ghostscript, and xdvi. I used a custom markup language to generate the printed and online versions of the thesis from the same source. The artifacts of my dissertation (including the markup language) are on my web home page, where anyone can download them, verify my results, and (I hope) improve them.

Chapter 1

Introduction

The world of pixel manipulation is a world of special cases. The need for rapid execution forces programmers to abandon the usual goals of modularity and reusability. The results are code bloat, inflexible systems, and missed deadlines. This thesis addresses the problem by looking at various graphics and media interfaces as programming languages, and using semantics-based techniques to transform flexible graphics operations into high-performance routines. More generally, we demonstrate a portable, easy-to-use interface to run-time code generation.

Programmers designing interfaces and implementing libraries with the C programming language expose alternate entry points or use fast-path optimization techniques[1] to support media with different numbers of bits per pixel, different alignments in memory, or optional channels such as transparency and depth. Unless a particular combination is directly implemented in one routine, its operation will require either conditional branches inside a loop over the pixels, or buffers to hold partial results. But conditional branches reduce bandwidth, and buffers increase latency and memory traffic. We can bypass these problems by generating special cases and fused, one-pass loops *as needed*. With this technique, known as run-time code generation (RTCG), code may depend on (and therefore must temporally follow) some program execution and input.

We face three questions. Can we produce good enough code? Is the time spent generating this code worth its speed-up? And is the programmer-time spent learning and using RTCG worth its speed-up? My dissertation examines the first question by building three prototype systems and measuring a collection of exam-

[1] With the fast-path technique, two implementations for a single operation are provided, one that is always correct, and one correct only on some condition, but that runs faster. The condition is checked every time the routine is invoked.

```
fun rgb_to_mono_1 rgb_start rgb_end m_start m_end =
    if (rgb_start = rgb_end) then ()
    else let val w = load_word rgb_start
         in (store_word m_start (76*(w&0xff) +
                                 154*((w>>8)&0xff) +
                                 25*((w>>16)&0xff));
             rgb_to_mono_1 (rgb_start+1) rgb_end
                           (m_start+1) m_end)
         end
```

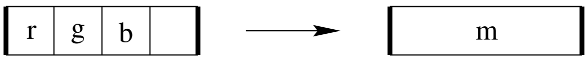

Figure 1.1: Color to monochrome conversion assuming RGBX interleaved input `rgb_start`, and 32-bit output `m_start`. The diagram below the code illustrates the memory layout of the source and destination buffers. Heavy lines indicate 32-bit word boundaries.

---

ples. The second question is addressed elsewhere. The third question remains for the future.

My thesis is that run-time code generation a programmer to write pixel-level graphics in a flexible high-level language without losing the performance of hand-specialized C. Furthermore, the notions of specialization, *binding times* (temporal types), and *compiler generation* from research in partial evaluation (PE) provide portable and accessible means to automate the creation of the special cases. The thesis is tested by building systems and benchmarking common kernels.

I claim that a programmer can write image-processing programs in an interpretive style where a program may include an image's layout in memory as well as operations on it. Such a program can be conceived, type-checked, and debugged as a normal one-stage program, then specialization can be used to compile these programs into efficient kernels.

Here is a concrete example. Say we need to convert a 24-bit RGB color image to grayscale. Two possible implementations appear in Figures 1.1 and 1.2. These and other examples use ML syntax extended with infix bit operations as found in the C programming language ($<<$ $>>$ `&` $|$). The `load_word` primitive accesses the contents of a memory location.

`Rgb_to_mono_1` assumes (1) that the pixels are stored one per 32-bit word

```
fun rgb_to_mono_2 r0 r1 b0 b1 g0 g1 m0 m1 =
    if (r0 = r1) then ()
    else let val rw = load_word r0
             val gw = load_word g0
             val bw = load_word b0
in
 (store_word m0
  (((9*(rw&0xff) + 20*(gw&0xff) + 3*(bw&0xff))>>5) |
   ((9*(rw>>8)&0xff + 20*(gw>>8)&0xff + 3*(bw>>8)&0xff)>>5)<<16);
  store_word (m0+1)
  (((9*(rw>>16)&0xff + 20*(gw>>16)&0xff + 3*(bw>>16)&0xff)>>5) |
   ((9*(rw>>24)&0xff + 20*(gw>>24)&0xff + 3*(bw>>24)&0xff)>>5)<<16);
  rgb_to_mono_2 (r0+1) r1 (b0+1) b1 (g0+1) g1 (m0+2) m1)
end
```

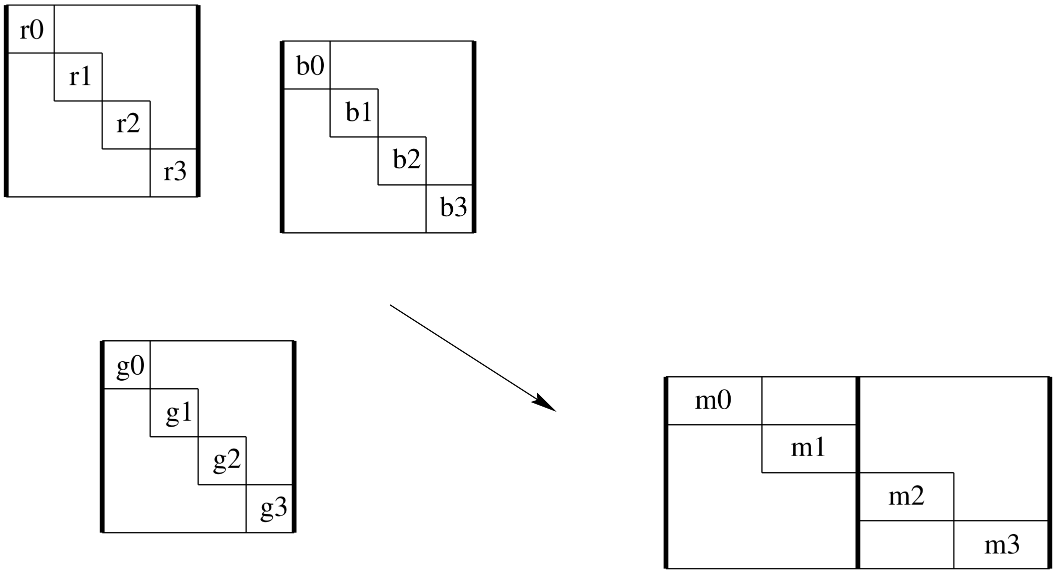

Figure 1.2: Color to monochrome assuming sequential input, 16-bit output, and a different combination function. Each row corresponds to an input pixel; this loop processes four pixels per iteration.

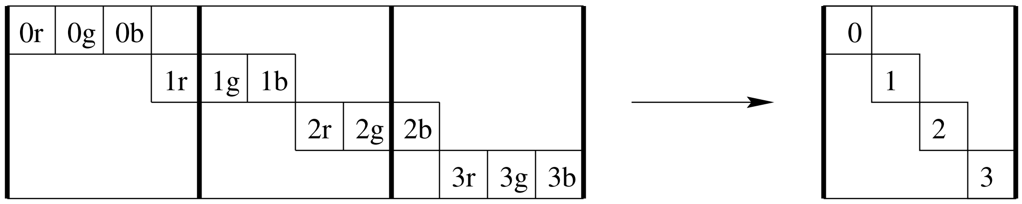

Figure 1.3: Another possible implementation packs four pixels into three words, saving 25% read-bandwidth over Figure 1.1. The code is similar to the code in Figure 1.2.

```
fun rgb_to_mono f r g b m =
  if (r s_end) then () else
    ((m s_put) (f (r s_get) (g s_get) (b s_get));
     rgb_to_mono f (r s_next) (g s_next)
                   (b s_next) (m s_next))
```

Figure 1.4: A signal-level implementation of color to monochrome conversion. The signals are represented with procedures that take a message (`s_put` etc).

(interleaved and ignoring 8 bits), (2) a particular linear weighting, and (3) 32-bit output. `Rgb_to_mono_2` assumes (1) that each channel is stored separately in memory (sequential) and word-aligned, (2) a different combination function, and (3) 16-bit word-aligned output. Other possibilites such as packing four pixels into three words (Figure 1.3), 12-bit resolution, or run-length coding would result in further variations. Codes like these make good use of instruction-level parallelism and run fast, but are of limited utility due to their assumptions.

A concrete result of this thesis is a system that can produce residual programs like Figures 1.1 and 1.2 from a general program that can handle any channel organization, bits per pixel, combination procedure, etc. Such a program appears in Figure 1.4.

In order to produce the fast special cases, I supply the assumptions (the program) to a code-generator (the compiler). Figure 1.5 shows how to do this. The quote syntax creates names for the equality assumptions rather than to name type variables (as in ML). Though slightly hypothetical, this language is essentially the Simple system from Section 5.3. Note that it has no ordinary (if any) type discipline. Section 1.2 summarizes how we build a procedure like `co_rgb_to_mono`

```
val rgbm1 =
    let fun S x = vector_signal ('start+x) ('stop+x) 32 8
        fun C x y z = 76*x + 154*y + 25*z
    in co_rgb_to_mono C (S 0) (S 8) (S 16)
                (vector_signal 'start1 'stop1 32 32)
    end

val rgbm2 =
    let fun S x y = vector_signal x y 8 8
        fun C x y z = (9*x+20*y+3*z)/32
    in co_rgb_to_mono C
         (S 'x0 'y0) (S 'x1 'y1) (S 'x2 'y2)
         (vector_signal 'start1 'stop1 16 16)
    end

rgbm1 start stop start1 stop1;
rgbm2 x0 y0 x1 y1 x2 y2 start1 stop1;
```

Figure 1.5: Syntax to create specialized versions from assumptions.

from the text of `rgb_to_mono`.

The rest of this chapter is organized as follows: the next section takes an abstract look at the software engineering problems of media processing and motivates my approach. Section 1.2 introduces the structures and techniques of specialization and partial evaluation. Section 1.3 connects this story to the published literature on the subject, and Section 1.4 identifies this work's novel contributions. The final section returns to the big picture and sketches some applications, and concludes by peering into the future.

1.1 Trade-offs

Programmers sometimes struggle with the contradictory goals of latency, bandwidth, and program size. This section explores three solution-paths to a series of increasingly general software design problems. Consider an audio-processing program that loops over many sound samples. Figure 1.6 gives pseudo-code for the natural implementation strategy. I start with a loop over a per-sample operation, then parameterize it by adding conditionals inside the loop.

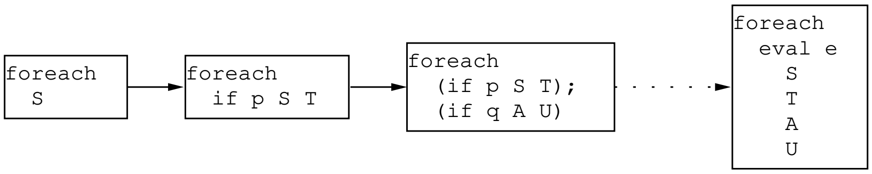

Figure 1.6: The interpretived alternative. `Foreach` loops over a statement; `eval` recursively switches over several statements. The leftmost box represents the unparameterized loop. Subsequent boxes represent parameterizations by booleans. The final box, connected with a dotted line, represents parameterization by an inductively defined type.

```
datatype Exp = e_val of int
             | e_var of string
             | e_plus of Exp * Exp
             | e_times of Exp * Exp
```

Figure 1.7: An inductively defined type that one might call a little language.

Eventually, the per-sample operation is parameterized with an inductively defined type. Such a parameter is sometimes called a *little language*, especially if the grammar is not regular and the language is Turing complete. Figure 1.7 contains an example. The per-sample operation has become a call to an interpreter for expressions in this language. Of course, the problem with this interpretive alternative is that these conditionals are tested repeatedly while their values remain constant. Although this kind of program runs very slowly, it is easy to write.

Another implementation appears in Figure 1.8. This technique is known as *buffering* or *batching*. The conditionals are independently hoisted out of the loops to eliminate the redundant tests. This requires multiple passes and temporary storage between passes. In the limit, this results in an interpreter with vector primitives. There are three problems with this alternative: accessing the buffers consumes memory bandwidth; the latency is increased because the first result is not ready until an entire buffer has been processed; and handling dynamic, sample-dependent control-flow becomes problematic.

We can think of these two alternatives as giving the inner control-flow to either the expression reduction or the samples. The third alternative appears in Figure 1.9; here we hoist the conditionals together, resulting in one loop for each combi-

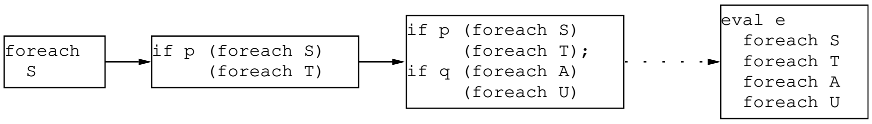

Figure 1.8: Buffered.

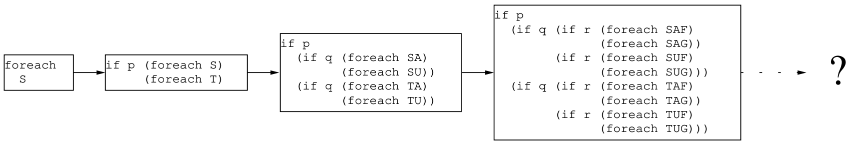

Figure 1.9: Specialized. The rightmost box has infinite size, so it is not depicted.

nation. Although this gives us optimal bandwidth and latency, because the number of combinations grows exponentially with the amount of parameterization, simple application of this alternative does not scale. In particular, when the type is inductive, the program would have infinite size.

The idea of run-time code generation is to avoid the exponential blow-up by generating the special cases lazily. By sacrificing a spike in latency we optimize asymptotic bandwidth and latency. The situation is depicted in the graphs of Figure 1.10.

When a problem is best solved with multiple passes, then the above arguments applie to the implementation of each pass. This happens in situations such as the compostion of wide convolutions.

Even if in theory my RTCG system can produce good enough code and can generate it fast enough, in reality it may be too difficult to use. After all, writing programs that write programs is a notoriously bug-prone process. The next section suggests an interface to RTCG designed to alleviate this problem.

1.2 Specialization

Specialization is a program transformation that takes a procedure and *some* of its arguments, and returns a procedure that is the special case obtained by fixing those arguments.

Formally, a specializer $[\![spec]\!]$ satisfies the following equation where f and $spec$ denote program texts, x and y denote ordinary values, and semantic brackets

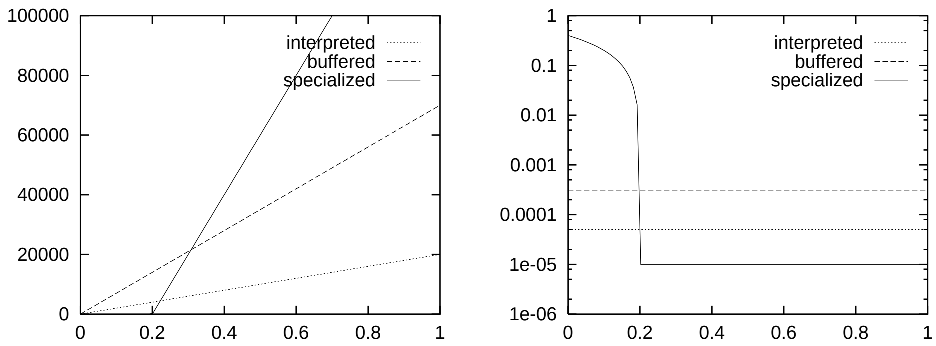

Figure 1.10: Throughput and latency of an idealized audio processor. The horizontal axes corresponds to time in seconds; on the left the verticle axis corresponds to total samples processed; on the right to the latency (how long until the currently received sample is processed) in seconds shown in logarithmic scale. We assume that all samples take the same amount of time to process.

$[\![\cdot]\!]$ denote ordinary evaluation:

$$[\![f]\!] \quad x \quad y \quad = \quad [\![\, [\![spec]\!] \quad f \quad x]\!] \quad y$$

There are many ways to implement $[\![spec]\!]$; a simple curry function suffices[2]. The intention is that $[\![spec]\!]$ will do as much work of f as is possible knowing only its first argument and return a *residual* program that finishes the computation. This gives us a way of factoring or staging computations [JoSche86] and is most useful if we use this residual program many times. In practice, specializers do less work than is possible in order to avoid code-space explosion. The annotations and heuristics used to decide when to stop working are the subject of Section 2.2.

Partial evaluation (PE) as described in [JoGoSe93, Consel88, WCRS91] is a syntax-directed, semantics-based, source-to-source program transformation that performs specialization. Although we say automatic, in fact some human input in the form of hints or annotations has proven necessary.

One of the primary applications of PE is compiler generation, frequently abbreviated *cogen*. If the function f above happens to be an interpreter, then $[\![spec]\!]$ f x is the compiled version of x, a program in the subject language. And so (at

[2]The ordinary curry function $\lambda fxy.f(x,y)$ maps functions to functions, but $[\![spec]\!]$ maps texts to texts. This distinction is obviated in a reflective system where the texts of functions can be recovered.

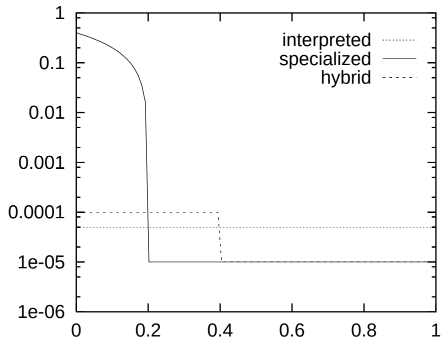

Figure 1.11: Latency of hybrid specialization and evaluation.

least theoretically) $[\![spec]\!]$ $spec\ f$ is a compiler for the language defined by the interpreter f. Another level of self-application yields $[\![spec]\!]$ $spec\ spec$, a compiler generator. These are known as the Futamura projections [Futamura71].

Research on practical compiler generation is widespread [Mosses78, JoSeSo85, Lee89]. This thesis concerns systems implemented directly with a static analysis known as binding-time analysis (BTA). Binding-time analysis classifies each variable and operation in the interpreter source text as either *static* (program) or *dynamic* (data). Basically, values that depend on dynamic values are dynamic. Recent research [DaPfe96] indicates that binding times can be modeled with temporal logic, and thus incorporated into type systems.

Let us now feel how compiler generation fits into the situation from the previous section. Initially the programmer type-checks and debugs a one-stage interpreter. Because specialization preserves the semantics of the code, producing an efficient two-stage procedure (a compiler that performs RTCG) is then just a matter of annotation and tweeking. Binding-time analysis performs the bookkeeping of program division and the specializer handles the mechanics of code construction. As an added bonus, since after annotation the source program can still run in one stage, it can be run in parallel with the compiler to soften the latency spike (Figure 1.11).

Cogen is not magic. It does not write any new code, it merely reorganizes the text of the procedures given to it and inserts calls to its own libraries. However, easing the creation of compilers from interpreters makes languages lightweight. Such a cogen promises to (and in fact may be specifically designed to) alleviate the implementation difficulties of interactive media.

```
(defvar g 2)

(defmacro f (x) `(+ g ,x))

(let ((g 17)) (f 419))
```

Figure 1.12: A Common LISP program that exhibits variable capture: the reference to `g` in the body of appears to refer to the global variable, but when expanded in the body of the `let`, the local definition of `g` takes precedence.

1.3 History

This section surveys the literature on metaprogramming and run-time code generation and places this work in perspective. This section uses a lot of jargon; explanations appear in the relevant references.

Run-time code generation *per se* has long history including exile, reconsideration [KeEgHe91], and a growing body of research demonstrating substantial performance gains in operating systems [PuMaIo88, EngKaOT95, MuVoMa97]. I define $\overline{\sigma}$ of an RTCG system as the average number of cycles spent by the compiler per instruction generation. For reference, a typical value of $\overline{\sigma}$ for a C or ML compiler is 175,000[3]

In the C and C++ programming world the lack of portable interfaces and the difficult nature of RTCG prevent more wide-spread use. However in the Common LISP world [Steele90], use of RTCG (via `eval`, `compile`, and `defmacro`) is considered an essential advanced technique [Graham94].

While macro-expansion happens at compile time rather than run time, it is the form of metaprogramming that people today are most familiar with. Experience with second-generation lexical macro systems such as ANSI C's [ANSI90] and the omission of macros from typed languages such as Java [GoJoSte96] and SML [MiToHa90] has given macro systems something of a bad reputation.

Despite this, s-expression systems continue to succeed with macros. Lisp's quasi-quote/unquote syntax was a good start, but syntactic systems suffer from the

[3] $\overline{\sigma}$ of GCC -O on x86 is about 200,000 (45 * 133M / (77k/2.5)), IRIX CC -O is about 120,000 (25 * 150M / (120k/4)), and SML/NJ v109.28 on a DEC Alpha is 176,000 (1320 * 150M / (4.5M/4)). The formula is the product of measured user time and clock speed divided by the number of instructions. The latter is estimated with the size of the compiler's output divided by the average number of bytes per instruction. Thanks to Perry Cheng for helping collect the SML data.

variable capture pitfall [KFFD86] (see Figure 1.12). Scheme's `syntax-rules` [R4RS] fixed this hygiene problem for a limited class of rewriting macros. Further improvements are the subject of active research [Carl96, HiDyBru92]. Macros for typed languages are also certainly possible [Haynes93]. One way to think about BTA is as a static analysis that places the backquotes and commas automatically.

The 'C language (pronounced "tick-C") [EnHsKa95] extends ANSI C with an interface for RTCG inspired by Lisp's backquote mechanism, though significantly more difficult to use due to limitations in the orthogonality, generality, and type regularity of the extensions. The recent implementation shows good performance in realistic situations with either of two backends [PoEnKa97]. `Tcc`'s ICODE backend performs basic optimizations such as instruction scheduling, peephole, and register allocation, resulting in $\overline{\sigma}$ between 1000 and 2500. The VCODE backend uses macros to emit code in one pass; $\overline{\sigma}$ is between 100 and 500.

Familiar compile-time systems for C include C++ templates [SteLe95] and parser generators such as yacc [Johnson75][4]. However, as corroborated by the work in compiler generation for C [CHNNV96, Andersen94], we believe C's lack of static semantics makes these systems inherently more difficult to build, use, and understand.

Fabius [LeLe96] uses fast automatic specialization for run-time code generation of a first-order subset of ML. Essentially, it is a compiler generator where the syntax of currying indicates staged computation, including memoization. Because the binding times are implicit in every function call, no inter-procedural analysis is required. Its extensions run very fast ($\overline{\sigma}$ is six).

Tempo [CHNNV96] attempts to automate RTCG for use by operating systems. It applies binding-time analysis combined with various other analyses to ANSI C. It emits GCC code that includes template-based code-generating functions, using a great hack to remain, at least in theory, as portable as GCC.

1.4 Contributions

This thesis makes the following contributions.

Nitrous is a directly-implemented memoizing compiler generator for a higher-order language. It accepts and produces (and its compilers produce) programs written in an intermediate language similar to three-address code. This design allows low-overhead run-time code generation as well as multi-stage compiler

[4]In fact, LR(k) parser generation is a special case of polyvariant compiler generation [Mossin93, SpeThi95].

generation (where one generates a compiler from an interpreter written in a language defined by another interpreter). This system is the subject of Chapter 4 and a previous paper [Draves96]. Its relevant features are:

Cyclic integers Standard PE systems have the ability to determine that `#1(s,d)` is static[5] where `s` is static and `d` is dynamic. The value `(s, d)` is called a partially-static structure. Nitrous supports a kind of partially static integer I call cyclic integers. With these, the BTA determines that `(d*32+5)%32` is static (five in this case).

Sharing/shapes Nitrous' compilers keep track of the names of the dynamic values. When one of these compilers generates code that moves a collection of values (such as procedure call/return), the compiler avoids generating code that moves multiple copies of the same value.

Conservative early equality Nitrous provides an operator that compares two dynamic values and returns a static result. The result is true if the compiler can prove that the values will be equal; false if they may not be equal.

A specializer with these features is powerful enough to implement (among other things) the subject of Chapter 3 and [Draves97]:

Bit-addressing This technique (one interface, one language) allows one to play with signal processing at the dataflow-level yet remain independent of the number of bits per sample. It uses a software implementation of a small fully associative cache. Because of the sharing analysis, the cache is classified as static and eliminated.

More generally, this was the first research to explicitly apply partial evaluation to run-time code generation [Draves95].

The rest of the dissertation consists of five chapters and an appendix. Chapter 2 defines a specializer $\mathcal{S}$ and briefly discusses its implementation, Chapter 3 extends it to cover bit-addressing, and Chapter 4 describes the Nitrous implementation. Chapter 5 presents benchmark data from Nitrous and from a small and simple implementation appear. Chapter 6 concludes by critically assessing the systems and considering how to improve them.

[5] In Lisp syntax that would read `(car (cons s d))`.

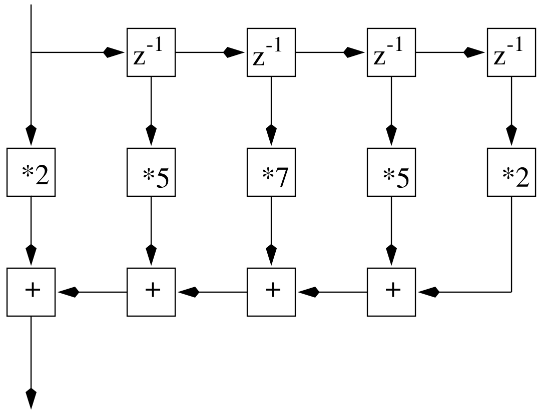

Figure 1.13: A linear finite filter built with tinker-toy DSP. A Box labeled z^{-1} is a delay; its value is the sample from the previous time slice.

```
val kernel = [2, 5, 7, 5, 2]
val prefix = ['a, 'b, 'c, 'd, 'e]
val filter5 = (filter sig16 kernel prefix)
```

Figure 1.14: A higher-level way to create the filter, using the same language as Figure 1.5

1.5 Applications and the Future

This section looks at what specializers can do for interactive media. We begin with some already implemented and benchmarked examples, then move on to speculation.

We can use bit-addressing to implement an object-oriented signal-processing interface. This is the language used in Figure 1.4 to write `rgb_to_mono`. It allows the programmer to work at the dataflow level, by connecting signals as if they were *tinker toys*. Figure 1.13 shows a linear filter built with a graphical transliteration of this interface. Figure 1.14 shows a higher-level way to build a filter. Section 5.3.1 discusses these systems.

As another example, consider an interactive sound designer. A particular voice is defined by a small program; Figure 1.15 is a typical example of a depiction

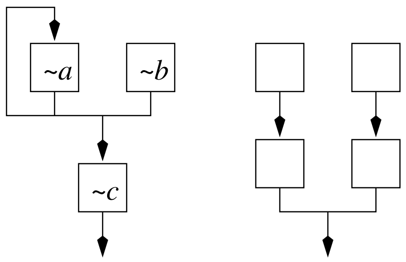

Figure 1.15: Two voices. On the left is a simple 2-in-1 synthesizer where oscillators *a* and *b* sum to modulate *c* as well as feeding back into *a*. On the right is another possibility.

---

of a wavetable synthesizer[6]. Most systems allow the user to pick from several predefined voices and adjust their scalar and wavetable parameters. With RTCG, the user may define voices with their own wiring diagrams.

Next example: consider a typical window system with graphics state consisting of the screen position of a window, the current video mode and resolution, the typeface, etc. Common graphics operations such as EraseRect, BitBlit, DrawString, and BrushStroke may be specialized to this graphics state. I expect that RTCG will increase perceived usability of systems when the number of graphics states in use at any one time is small relative to the number of potentially useful states, and the time spent doing graphics operations is large.

Operations such as decoding an image and blitting it to the screen are ordinarily implemented in two-passes. With RTCG, when a file or network connection to a compressed video source is opened, a DecodeAndBlit routine may be generated that avoids an intermediary buffer, and thus reduces communication latency. The same idea applies to the parts of an operating system that implement network protocols.

My final example: artificial evolution of two-dimensional cellular automata. The standard technique is to apply the genetic algorithm to lookup-tables indexed by all possible neighborhoods [MiHraCru94]. But if the cells have just three bits of state and a 3-by-3 neighborhood then the lookup-table would require 192 Mbytes ($3{\cdot}2^{3\cdot 9}$ bits). With RTCG one can mutate data-structures describing programs (i.e. substitute genetic programming for the genetic algorithm), and then

[6]Wavetable synthesis is just like FM (Frequency Modulation) synthesis, but sinusoids are replaced with lookup-tables.

synthesize loops that apply these area-operations in efficient (tiled or striped) order.

In summary I believe that metaprogramming in the form of run-time code generation can have significant impact on signal-processing and graphics applications. Specialization promises to give us a practical interface to RTCG. With this we can build systems with high bandwidth and low latency without giving up flexibility. I believe such systems will be very important in a future where our personal communications are mediated by computer networks.

Chapter 2

Specialization

Section 1.2 defined the behavior of a specializer $[\![\,spec\,]\!]$ with

$[\![\,f\,]\!]\ \ x\ \ y\ =\ [\![\,[\![\,spec\,]\!]\ \ f\ \ x\,]\!]\ \ y$

A specializer produces special cases from general procedures. This chapter exhibits one such system ($\mathcal{S}$), and uses it to understand the practice of polyvariant specialization. First I introduce my notation and give a simple online specializer for a typed λ-language. I then extend the calculus with products, booleans, and sums. The remaining sections discuss lifting, memoization, recursion, and compiler generation. The correctness of $\mathcal{S}$ is considered in Section 3.5.

This chapter is generally a review of Partial Evaluation (PE) practice; [CoDa98] and [JoGoSe93] are the standard texts of the field and may be considered references of first resort. [Jones91] is a theoretical introduction to operational behavior of specializers, and [Jones88] provides a practical description of first-order PE. [WCRS91, Consel88, Thiemann96, BoDa91, BiWe93] are system descriptions.

Figure 2.1 gives the grammar of the object language and its type structure. It also defines some domains [Reynolds97] and their associated metavariables. The language is a call-by-value λ-calculus extended with integer constants, primitives, conditionals, a lift annotation, and explicit types on abstractions. I use the `typewriter` face for the terms of the object language. I use $\diamond$ to denote a generic, "black box" binary primitive operation. $\boxed{\text{Frames}}$ mark manipulation of the terms of the λ-language's syntax and types. Frames appear in patterns and expressions. The slots of the term in the frame are either metavariables or parenthesized expressions in the metalanguage.

Figure 2.2 gives a specializer $\mathcal{S}$. Roman typeface is used for metalanguage constructs such as "let pat=e in e" and "if e then e else e". "Match $e\ pat\ \rightarrow\ e$

..." denotes pattern matching where the metavariables only match members of the appropriate domain. I use $[v \mapsto m]\rho$ to denote the extension of the environment ρ with a binding from the variable v to the value m, and [] to denote the empty environment.

Note that several operators have two notations, one in the metalanguage and one in the object language. For example `lambda` is the syntax for a procedure and λ is a mathematical function; `*` denotes the product type and $\times$ denotes the product of two domains.

$\mathcal{S}$ is a *partial-evaluation* function, it assigns a meaning from M to a source text with environment. The difference from the ordinary semantics is that M contains Exp, whose members represent computations dependent on unknown values, that is, residual code. I say the specializer *emits* residual code.

The `lift` primitive forces its argument to become residual code; I call it an annotation because it has no meaning in the ordinary, one-stage semantics.

Figure 2.3 defines the reflection and reification functions U and D (up from the subject language and down from the metalanguage). They operate as coercions between code and data; understanding them is not essential to this chapter. D is invoked in the term for the meaning of `lift`.

An important property that this specializer lacks is safety. $\mathcal{S}$ may lose effects or duplicate computations. For example, the rule for static function application works by substitution, so if the computation of an argument has a side-effect, but is then passed to a function that never refers to the value, the effect is lost. Similarly, if the value is used more than once, the effect may be duplicated.

Most systems use let-insertion [BoDa91] to guarantee safety. The implementation of let-insertion is closely related to that of booleans, explained below [Bondorf92, Danvy96].

Note that the `if0` clause requires that when a conditional has dynamic predicate, then both arms are also dynamic. Section 2.1 below shows how to implement a conditional without this restriction.

$\mathcal{S}$ is similar to the λ-mix of [GoJo91], but λ-mix uses uses a two-level input language where source λs have been labeled either for execution or immediate residualization. $\mathcal{S}$ reserves judgment until the function is applied; $\mathcal{S}$ depends on a lift annotations to emit a `lambda`.

Note that many cases are missing from $\mathcal{S}$. I assume that all input programs are type-correct and lift annotations appear as necessary, so these situations never occur. Placement of the lifts is crucial to successful staging: too many lifts and $\mathcal{S}$ degenerates into the curry function; too few and $\mathcal{S}$ fails to terminate. Typically

$$t \in \mathsf{Type} ::= \texttt{int} \mid \mathsf{Type} \texttt{ -> } \mathsf{Type}$$

$$\begin{aligned} d, e \in \mathsf{Exp} ::= {} & \mathsf{Atom} \mid \mathsf{Var} \mid \texttt{lambda}\ \ \mathsf{Var}\,\texttt{:}\,\mathsf{Type}\,\texttt{.}\,\mathsf{Exp} \\ & \mid \mathsf{Exp}\ \mathsf{Exp} \mid \texttt{if0}\ \mathsf{Exp}\ \mathsf{Exp}\ \mathsf{Exp} \\ & \mid \texttt{lift}\ \mathsf{Exp} \mid \mathsf{Exp} \diamond \mathsf{Exp} \mid \ldots \end{aligned}$$

$$\begin{aligned} & s, k \in \mathsf{Atom} = \mathbb{Z} \\ & f_t \in \mathsf{F} = (\mathsf{M} \to \mathsf{M}) \times \mathsf{Type} \\ & m \in \mathsf{M} = \mathsf{Exp} + \mathsf{Atom}_\perp + \mathsf{F} \\ & \rho \in \mathsf{Env} = \mathsf{Var} \to \mathsf{M} \end{aligned}$$

Figure 2.1: The λ-language, types, domains, and metavariables.

$$\mathcal{S} : \mathsf{Exp} \to \mathsf{Env} \to \mathsf{M}$$

$$\begin{aligned}
& \mathcal{S}\ \boxed{e_0 \diamond e_1}\ \rho = \text{match } (\mathcal{S}\ e_0\ \rho, \mathcal{S}\ e_1\ \rho) \\
& \qquad\qquad (s_0, s_1) \to s_0 \diamond s_1 \\
& \qquad\qquad (d_0, d_1) \to \boxed{d_0 \diamond d_1} \\
& \mathcal{S}\ \boxed{v}\ \rho = \rho\ v \\
& \mathcal{S}\ \boxed{k}\ \rho = k \\
& \mathcal{S}\ \boxed{\texttt{lambda}\ \ v\,\texttt{:}\,t\,\texttt{.}\,e}\ \rho = (\lambda v'.\mathcal{S}\ e\ ([v \mapsto v']\rho))_t \\
& \mathcal{S}\ \boxed{e_0\ e_1}\ \rho = \text{match } (\mathcal{S}\ e_0\ \rho, \mathcal{S}\ e_1\ \rho) \\
& \qquad\qquad (f, m) \to f\ m \\
& \qquad\qquad (d_0, d_1) \to \boxed{d_0\ d_1} \\
& \mathcal{S}\ \boxed{\texttt{lift}\ \ e}\ \rho = D\ (\mathcal{S}\ e\ \rho) \\
& \mathcal{S}\ \boxed{\texttt{if0}\ \ e_0\ \ e_1\ \ e_2}\ \rho = \text{match } (\mathcal{S}\ e_0\ \rho) \\
& \qquad\qquad s_0 \to \text{if } 0 = s_0 \text{ then } (\mathcal{S}\ e_1\ \rho) \text{ else } (\mathcal{S}\ e_2\ \rho) \\
& \qquad\qquad d_0 \to \text{let } d_1 = \mathcal{S}\ e_1\ \rho \\
& \qquad\qquad\qquad\quad d_2 = \mathcal{S}\ e_2\ \rho \\
& \qquad\qquad\qquad \text{in } \boxed{\texttt{if0}\ \ d_0\ \ d_1\ \ d_2}
\end{aligned}$$

Figure 2.2: A direct-style specializer. The apparently missing cases are not needed because input programs are assumed to be correctly annotated.

$$
\begin{aligned}
&D : \mathsf{M} \rightarrow \mathsf{Exp}\\
&D\ d = d\\
&D\ s = \boxed{s}\\
&D\ f_t = \text{let } v' = \text{gensym}\\
&\qquad\qquad e' = D(f(Utv'))\\
&\qquad\quad \text{in} \boxed{\texttt{lambda } v'\texttt{:}t\texttt{.}e'}
\end{aligned}
$$

$$
\begin{aligned}
&U : \mathsf{Type} \rightarrow \mathsf{Exp} \rightarrow \mathsf{M}\\
&U \boxed{t\texttt{->}t'}\ e_0 = (\lambda\ v\ .\ \text{let } e_1 = D\ v \text{ in } U\ t \boxed{e_0\ e_1})_t\\
&U\ \texttt{int} \boxed{e} = e
\end{aligned}
$$

Figure 2.3: Reification function D and reflection function U. Gensym makes a fresh variable.

```
fun tail_loop b e r =
  if e = 0 then r
  else tail_loop b (e-1) (b*r)

val power b e = tail_loop b e 1
```

Figure 2.4: Specialization of a tail-recursive function may not terminate without annotation.

binding time analysis (BTA) is combined with programmer annotations to insert the lifts. For example, if ρ=[a$\mapsto$6 b$\mapsto\boxed{\texttt{c}}$] then $\mathcal{S}$ requires `((lift a)`$\diamond$`b)` rather than `(a`$\diamond$`b)`. This kind of lift is obvious, and is easily handled by BTA. As an example of the kind of lift that cannot be easily automated, consider the tail-recursive function in Figure 2.4, where `e` is dynamic and `b` is static. Unless we manually lift `r` to dynamic, $\mathcal{S}$ will diverge.

If each variable and procedure always has the same binding time, then a partial evaluator is said to be *monovariant* in binding times. The prototypical monovariant system is λ-mix. Monovariant BTA is well-understood and can be efficiently implemented with type-inference [Henglein91]. The problem with this kind of system is that frequently a procedure is applied to values with different binding times. For example, in one context I might apply `power` to static base and exponent, but in another, to dynamic base and static exponent.

Type ::= ... | Type `*` Type
M = ... + (M × M)
Exp ::= ... | Exp`,` Exp | `L` Exp | `R` Exp

$\mathcal{S}\ \boxed{e_0\texttt{,}\ \ e_1}\ \rho = (\mathcal{S}\ e_0\ \rho), (\mathcal{S}\ e_1\ \rho)$
$\mathcal{S}\ \boxed{\texttt{L}\ \ e}\ \rho = \text{match}\ (\mathcal{S}\ e\ \rho)\ (m_0, m_1) \to m_0$
$\mathcal{S}\ \boxed{\texttt{R}\ \ e}\ \rho = \text{match}\ (\mathcal{S}\ e\ \rho)\ (m_0, m_1) \to m_1$

$D\ (m_0, m_1) = \boxed{(D\ m_0)\texttt{,}\ (D\ m_1)}$
$U\ \boxed{t\ \ \texttt{*}\ \ t'}\ e = (U\ t\ \boxed{\texttt{L}\ \ e}), (U\ t'\ \boxed{\texttt{R}\ \ e})$

Figure 2.5: Extensions for product types. A comma denotes a pair construction.

If the binding times are context sensitive, then a partial evaluator is said to be polyvariant in binding times. A polyvariant BTA is one that effectively places lifts for this kind of system. Polyvariant BTA usually implemented with abstract interpretation [Consel93]. Thus a given piece of syntax may be both executed by $\mathcal{S}$ and emitted as residual.

2.1 Products and Sums

This section adds product and sum types to the specializer. Inductively defined types are similarly easy, but other types such as polymorphism and modules are the subject of active research.

The specializer above treats primitives uniformly. A primitive application is either performed at specialization time or emitted as residual code. In particular, if a primitive has one static and one dynamic argument, then the static one must be lifted, throwing information out of the compiler. But many primitives have algebraic properties that allow us to preserve some information.

Figures 2.5, 2.6, and 2.7 extend the formal system to products, booleans, and sums, respectively. Sums are a generalization of booleans as Bool = Unit + Unit.

The product types allow the following:

$\mathcal{S}$ `(L(0,d))` $\rho = 0$

where ρ=[d$\mapsto$$\boxed{\texttt{d}}$]. In partial evaluation jargon, the value of `(0,d)` is called a *partially static structure*.

This `if` and `if0` from Figure 2.2 are fundamentally different:

$$
\begin{aligned}
\mathcal{S}(\texttt{1 + (if0 d 2 3)})\rho &= \texttt{1 + (if0 d 2 3)} \\
\mathcal{S}(\texttt{1 + (if d 2 3)})\rho &= \texttt{(if d 3 4)}
\end{aligned}
$$

Normally, we think of specialization as either performing or emitting each operation of a program. But this specialization of `if` requires that the addition be performed twice. How can $\mathcal{S}$ return into the addition twice twice?

There are two standard solutions: continuation-based specialization [Bondorf92], and the shift and reset control operators used here. Shift is similar to Scheme's `call/cc` (call with current continuation), but the extent of κ is limited by reset. See Section A.8 for a explanation of these control operators.

Section 4.2 shows how to get the same result by using a source language in continuation-passing style.

2.2 Lifting

We originally said that a specializer does "all the work possible given only some of the input". The previous section gave rules to find and perform this work. The problem is that blind application of these rules often finds too much work, and creates many specialized procedures that do not run fast enough to pay for their size and number. In particular, since the rule for function application may inline, use of recursion may result in infinite unfolding, and thus non-termination of the specializer.

The `lift` term is the fundamental form of control, but is insufficient. For example, in the tail-recursive `power` function in Figure 2.4 we must lift `r` to prevent non-termination. But if `e` is static, then lifting `r` prevents the simple static result we surely want.

Nitrous and Schism use lift languages to give the programmer conditional lifting. Basically they work by reifying binding times. Other systems such as [WCRS91] contain analyses that identify and perform this lift, but miss-identify others. I believe that full automation (i.e. elimination of `lift` from the input language) is not yet feasible; binding-time and staging systems need some kind of manual control.

Type ::= ... | `bool`
M = ... + Bool
Exp ::= ... | `if` Exp Exp Exp

$$
\begin{aligned}
\mathcal{S}\ \boxed{\texttt{if}\ e_0\ \ e_1\ \ e_2}\ \rho = \text{match}\ (\mathcal{S}\ e_0\ \rho)\quad & \\
s_0 &\to \text{if } s_0 \text{ then } (\mathcal{S}\ e_1\ \rho) \text{ else } (\mathcal{S}\ e_2\ \rho) \\
d_0 &\to \text{let } d_1 = \mathcal{S}\ e_1\ \rho \\
&\qquad\quad d_2 = \mathcal{S}\ e_2\ \rho \\
&\qquad \text{in } \boxed{\texttt{if}\ d_0\ \ d_1\ \ d_2}
\end{aligned}
$$

$$
\begin{aligned}
D\ f_t = \text{let } & v' = \text{gensym} \\
& e' = \text{reset } D(f(Utv')) \\
\text{in } & \boxed{\texttt{lambda}\ v'\texttt{:}t\texttt{.}e'}
\end{aligned}
$$

$$
\begin{aligned}
U\ \texttt{bool}\ e = \text{shift}\ \kappa\ \text{in} & \\
\text{let } & e_0 = \text{reset } (\kappa \text{ true}) \\
& e_1 = \text{reset } (\kappa \text{ false}) \\
\text{in } & \boxed{\texttt{if}\ e\ \ e_0\ \ e_1}
\end{aligned}
$$

Figure 2.6: Extensions for booleans, including replacement function reifier.

$$
\begin{array}{l}
\mathsf{Type} ::= \ldots \mid \mathsf{Type} + \mathsf{Type} \\
\mathsf{M} = \ldots + (\text{inL}\ \mathsf{M}) + (\text{inR}\ \mathsf{M}) \\
\mathsf{Exp} ::= \ldots \mid \mathtt{inL}\ \mathsf{Exp} \mid \mathtt{inR}\ \mathsf{Exp} \\
\qquad\quad \mid \mathtt{case}\ \mathsf{Exp}\ \mathsf{Var}\ \mathsf{Exp}\ \mathsf{Var}\ \mathsf{Exp}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{S}\ \boxed{\mathtt{inL}\ e}\ \rho = \text{inL}\ (\mathcal{S}\ e\ \rho) \\
\mathcal{S}\ \boxed{\mathtt{inR}\ e}\ \rho = \text{inR}\ (\mathcal{S}\ e\ \rho)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{S}\ \boxed{\mathtt{case}\ e\ v_0\ e_0\ v_1\ e_1}\ \rho = \text{match}\ \mathcal{S}\ e\ \rho \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{inL}\ m_0 \rightarrow \mathcal{S}\ e_0\ ([v_0 \mapsto m_0]\ \rho) \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{inR}\ m_1 \rightarrow \mathcal{S}\ e_1\ ([v_1 \mapsto m_1]\ \rho)
\end{array}
$$

$$
\begin{array}{l}
D\ (\text{inL}\ m_0) = \boxed{\mathtt{inL}\ \mathtt{(}D\ \ m_0\mathtt{)}} \\
D\ (\text{inR}\ m_0) = \boxed{\mathtt{inR}\ \mathtt{(}D\ \ m_0\mathtt{)}}
\end{array}
$$

$$
\begin{array}{l}
U\ \boxed{t + t'}\ e = \text{shift}\ \kappa\ \text{in} \\
\qquad\qquad \text{let}\ v_0 = \text{gensym} \\
\qquad\qquad\quad\ \ v_1 = \text{gensym} \\
\qquad\qquad\quad\ \ e_0 = \text{reset}\ (\kappa\ (\text{inL}\ (U\ t\ v_0))) \\
\qquad\qquad\quad\ \ e_1 = \text{reset}\ (\kappa\ (\text{inR}\ (U\ t'\ v_1))) \\
\qquad\qquad \text{in}\ \boxed{\mathtt{case}\ e\ v_0\ e_0\ v_1\ e_1}
\end{array}
$$

Figure 2.7: Extensions for sum types.

```
(define (power-abs n)
  (lambda (*)
    (lambda (a)
      (fix1 (lambda (loop)
              (lambda (n)
                (if (zero? n)
                    1
                    (* a (loop (- n 1))))))))))

(define-base-type Int "i")
(define-compound-type Times ((Int Int) => Int) "*" alias)
(define (fix1 F) (lambda (x) ((F (fix1 F)) x)))
(residualize (power-abs 5) '(Times -> Int -> Int))
```

$\rightarrow$

```
(lambda (*)
  (lambda (i0)
    (* i0 (* i0 (* i0 (* i0 (* i0 1)))))))
```

Figure 2.8: TDPE of power, static exponent. The double arrow denotes a multi-argument function type.

2.3 Recursion and Memoization

Memoization is a standard technique to avoid repeated computation. It plays an important role in some specializers, including ours. This section explains the distinction by looking at dynamic loops. We call specializers that use this kind of memoization *polyvariant*, and those that do not *monovariant*.

Using the rules in Figure 2.2 above, a loop must be built with a fixed-point operator (this is one of the things that the ML syntax obscures). The type of the operator depends on how much of the recursion is to be expanded. In other words, the source program must be adjusted to support different binding times. Figures 2.8 and 2.9 show two versions of the power function specialized with Type Directed Partial Evaluation (TDPE) [Danvy96][1]. A polyvariant specializer does not have this limitation, as the example in Figure 2.10 demonstrates.

At a call site, a monovariant specializer like λ-mix [GoJo91] either inlines or residualizes. A polyvariant specializer has the option of emitting a call to a

[1]Danvy reports that a future version of TDPE does not have this restriction.

```
(define (dpower-abs fix2 * zero? -)
  (fix2 (lambda (loop)
          (lambda (x n)
            (if (zero? n)
                1
                (* x (loop x (- n 1))))))))

(define-base-type Bool "b")
(define-compound-type Minus ((Int Int) => Int) "-" alias)
(define-compound-type Zero? (Int -> Bool) "zero?" alias)

(define-compound-type Fix2
  ((((Int Int) => Int) -> ((Int Int) => Int)) ->
   ((Int Int) => Int))
  "fix2" alias)

(define (fix2 F) (lambda (x y) ((F (fix2 F)) x y)))

(residualize dpower-abs
  '((Fix2 Times Zero? Minus) => (Int Int) => Int))
```

$\rightarrow$

```
(lambda (fix2 * zero? -)
  (lambda (i3 i4)
    ((fix2 (lambda (x0)
             (lambda (i1 i2)
               (if (zero? i2)
                   1
                   (* i1 (x0 i1 (- i2 1)))))))
     i3 i4)))
```

Figure 2.9: TDPE of power, fully dynamic version.

```
(similix 'power (list '*** 5) "source.sim")
→
(define (power5 b)
  (* b (* b (* b (* b (* b 1))))))


(similix 'power (list '*** '***) "source.sim")
→
(define (power-dd b e)
  (if (= e 0)
      1
      (* b (power-dd b (- e 1)))))
```

Figure 2.10: Specialization of power with Similix. `***` indicates a dynamic value, the contents of `source.sim` appear in Figure 2.11.

specialized procedure defined elsewhere, and passing just the dynamic arguments. This specialized procedure may be called from several different places.

A polyvariant specializer uses its memo-table to *discover* dynamic fixed points. Before creating a specialized version of a procedure, a polyvariant system checks to see if it already has a version of that procedure specialized to those static values, and if so, creates a call to the already defined procedure.

This use of memoization is more than just a convenience or a simple performance improvement. Polyvariance is fundamentally more powerful because zero, one, or more fixed points may be created from a single source fixed point. Furthermore, the types of the residual fixed-points may be different from the types of the input fixed points. The standard example is Ackermann's function specialized to its first argument, resulting in two different fixed points, as in Figure 2.12. Figure 2.13 shows an example of a loop that unrolls several times before the static values match and the dynamic loop is found. A more practical example is the generation of KMP string-matching by specialization [CoDa89]. None of these results can be duplicated with a monovariant system. It can be dangerous, however, to rely on the memo-table to find fixed-points.

The question remains: when should the system inline, and when should it specialize, that is, test the memo-table and form a call? While it is safe to always specialize, the resulting code has many trivial calls. Many systems, including ours, use the *dynamic-conditional heuristic* [BoDa91]. The heuristic suggests that

```
(define (power b e)
  (if (= e 0)
      1
      (* b (power b (- e 1)))))

(define (ack m n)
  (cond ((zero? m) (+ n 1))
        ((zero? n) (ack (- m 1) 1))
        (else (ack (- m 1) (ack m (- n 1))))))

(define (mize n m d e)
  (if (e) n
      (let* ((n1 (- n d))
             (n2 (if (< n1 0) (+ n1 m) n1)))
      (mize n2 m d e))))
```

Figure 2.11: The contents of the file `source.sim`

```
(similix 'ack (list 2 '***)  "source.sim")
→
(define (ack2 n)
  (if (zero? n)
      (ack1 1)
      (ack1 (ack2 (- n 1)))))
(define (ack1 n)
  (if (zero? n)
      2
      (+ (ack1 (- n 1)) 1)))
```

Figure 2.12: Ackermann's function specialized with Similix.

```
(similix 'mize (list 0 5 2 '***) "source.sim")
→
(define (mize052 e)
  (cond ((e) 0) ((e) 3)
        ((e) 1) ((e) 4)
        ((e) 2)
        (else (mize052 e))))
```

Figure 2.13: Memoization in action.

calls be inlined unless the body of the procedure contains a control-flow branch depending on dynamic values. Such a branch is known as a dynamic conditional. The intuition behind the heuristic is that if there is no dynamic conditional then inlining is safe because any resulting non-termination would also be present in ordinary execution of the source program[2].

The price is that polyvariant specialization is more difficult to implement, type, and reason about. Bit-addressing makes essential use of memoization.

2.4 Compiler Generation

If we used a literal implementation of $\mathcal{S}$ to specialize programs, then every time we generate a residual program, we would also traverse and dispatch on the source text. The standard way to avoid this repeated work is to introduce another stage of computation. That is, to use a compiler generator *cogen* instead of a specializer *spec*. The compiler generator converts f into a synthesizer of specialized versions of f:

$$[\![f]\!] \;\; x \;\; y \;=\; [\![\, [\![\, [\![cogen]\!] \;\; f]\!] \;\; x]\!] \;\; y$$

These systems are called compiler generators because if f is an interpreter, then $[\![cogen]\!] \, f$ is a compiler.

[2]This heuristic is very simple and aggressive. This kind of manual control of inlining sometimes produces unexpectedly large results. Compilers for functional languages such as SML [ApMa91] and Haskell [JHHPW93] make much more sophisticated inlining decisions. One could use the safe, always-specialize heuristic, and use sophisticated inlining as a post-pass to remove the trivial calls. In the run-time environment we are interested in, feedback from execution or a inexpensive approximation may substitute for sophisticated inlining decisions.

The standard way of implementing a compiler generator begins with a static analysis of the program text, then produces the synthesizer by syntax-directed traversal of the text annotated with the results of the analysis. Cogen knows what will be constant but not the constants themselves. We refer to this information *binding times*, it corresponds to the injection tags on members of M. We say members of Atom are *static* and members of Exp are *dynamic*. The binding times form a lattice because they represent partial information. It is always safe for the compiler to throw away information; this is called *lifting* and is the source of the `lift` annotation in the λ-language. Thus the bottom of the lattice is dynamic.

[BoDu93] shows how to derive a cogen from λ-mix in two steps. The first step converts a specializer into a compiler generator by adding an extra level of quoting to $\mathcal{S}$ so static statements are copied into the compiler and dynamic ones are emitted. The second step involves adding a continuation argument to $\mathcal{S}$ to allow propagation of a static context into the arms of a conditional with a dynamic test. One of the interesting results of [Danvy96] is how this property (the handling of sum-types and let-insertion) can be achieved while remaining in direct style by using shift/reset.

A remarkably pleasing though seemingly less practical way of implementing $[\![cogen]\!]$ is by self-application of a specializer $[\![\, [\![mix]\!] \;\; mix \;\; mix \,]\!]$, as suggested in [Futamura71] and first implemented in [JoSeSo85].

Specializers are classified as either *offline* or *online*. An online system works in one pass. Roughly, offline systems are suitable for compiler generation via self-application, and online systems are not. An offline system works in two phases. First it performs a whole-program analysis phase (the BTA) independent of the static values, and then it performs a specialization phase. Ideally, in an offline system all perform-vs-emit decisions are made in the BTA, but in reality most offline systems include polyvariance, which involves value-dependent decisions. Further hybridization is explored in [Sperber96]. The system from chapter 4 memoizes on binding times and static values, so it is also a hybrid. As a result, procedure contents are treated offline, but procedure calls are treated online. The system in Section 5.3 is online. Similix (see Section A.7) is offline.

2.5 Summary

This chapter covers the theory and practice of specialization. In order to understand the next chapter, it is important to understand that the specializer performs constant propagation, function inlining, and memoization.

Chapter 3

Bit-addressing

Media such as audio, images, and video are increasingly common in computer systems. Such data are represented by large arrays of small integers known as *samples*. Rather than wasting bits, samples are packed into memory. Figure 3.1 illustrates three examples: monaural sound stored as an array of 16-bit values, a grayscale image stored as an array of 8-bit values, and 8-bit values compressed with run-length encoding. Figure 1.3 shows a color image stored as interleaved 8-bit arrays of red, green, and blue samples. Such ordered collections of samples are called *signals*. A constant signal may avoid memory usage completely.

We begin by considering only signals that are regular arrays. Say we specify a signal's representation with four integers: `from` and `to` are bit addresses; `size` and `stride` are numbers of bits. The signal itself is stored in memory as a array where the distance (in bits) between elements is the stride, and the number of bits per element is the size. I use little-endian addressing so the least significant bit of each word (LSB) has the least address of the bits in that word.

```
type baddress = int
type signal = baddress * baddress * int * int
           (* from       to         size  stride *)
```

Figure 3.2 gives the code to sum the elements of such a signal. As in Chapter 1, the examples use ML syntax extended with infix bit operations and a `load_word` primitive. This chapter assumes 32-bit words, but any other size could just as easily be substituted, even at run-time. The integer division (`/`) rounds toward minus infinity, and the integer remainder (`%`) has positive base and non-negative result. To simplify this presentation, `load_sample` does not handle samples that cross word boundaries. This procedure is the beginning of the bit-level abstraction.

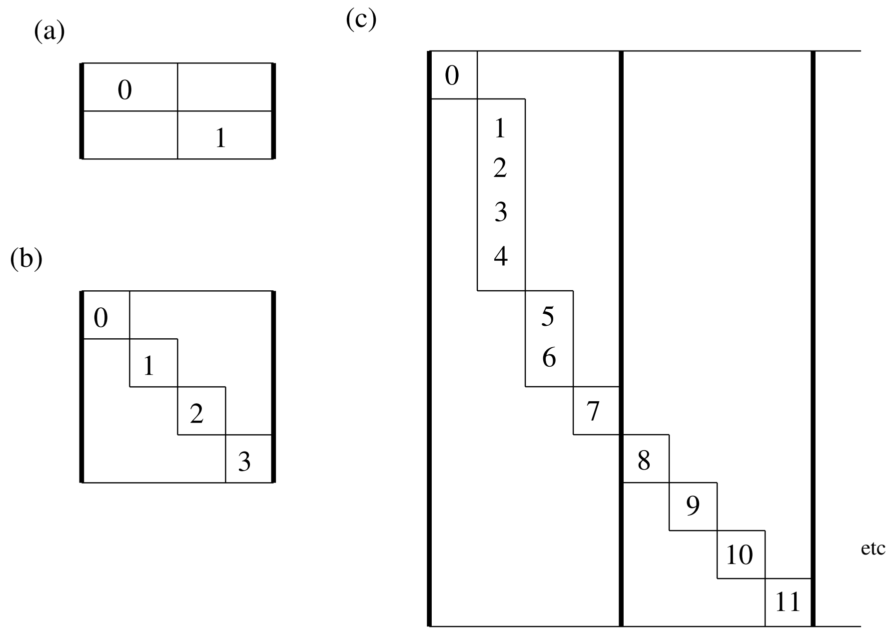

Figure 3.1: Layout of (a) 16-bit monaural sound, (b) an 8-bit grayscale image, and (c) a run-length encoded array where multiple samples occupy the same location in memory. The heavy lines indicate 32-bit word boundaries.

```
fun sum (from, to, size, stride) r =
  if from = to then r else
  sum ((from + stride), to, size, stride)
      (r + (load_sample from size))

fun load_sample p b =
   ((1 << b) - 1) &
   ((load_word (p / 32)) >> (p % 32))
```

Figure 3.2: Summing a signal using bit addressing. Use of equality instead of greater/less-than for the end-test is essential to support negative strides.

```
fun sum_0088 from to r =
  if from = to then r else
  let val v = load_word from
  in sum_0088 (from + 1) to
    (r + (v & 255) + ((v >> 8) & 255) +
    ((v >> 16) & 255)+ ((v >> 24) & 255))
  end
```

Figure 3.3: Summing a signal assuming packed, aligned 8-bit samples as in Figure 3.1(b).

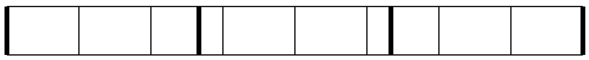

Figure 3.4: 12-bit signal against 32-bit words shown with abbreviated vertical axis.

If I fix the layout of the input to `sum` by assuming that

1. `stride = size = 8` and
2. `(from % 32) = (to % 32) = 0`

then the implementation in Figure 3.3 computes the same value, but runs more than five times faster (see Figure 5.1). There are many reasons: the loop is unrolled four times, resulting in fewer conditionals and more instruction-level parallelism. The shift offsets and masks are known statically, allowing immediate-mode instruction selection. The division and remainder computations are avoided, and redundant loads are eliminated.

Different assumptions result in different code. For example, sequential 12-bit samples result in unrolling 8=lcm(12,32)/12 times so that three whole words are loaded each iteration (see Figure 3.4). Handling samples that cross word boundaries requires adding a conditional to `load_sample` that loads an additional word, then does a shift-mask-shift-or sequence of operations. The actual implementation appears in Appendix A.1.

This chapter shows how to use a specializer to derive code like Figure 3.3 from the code in Figure 3.2 automatically. First I introduce *cyclic integers*, which provide intelligent unrolling. Next, I show how to implement a cache in software to optimize loads and stores. Finally some limitations of this approach are revealed.

$\langle b\ q\ r \rangle \in \mathsf{Cyclic} = \mathbb{Z} \times \mathsf{Exp} \times \mathbb{Z}$

$m \in \mathsf{M} = \ldots + \mathsf{Cyclic}$

$D\ \langle b\ q\ r \rangle = \boxed{b\star q+r}$

Figure 3.5: Extending domains and D for cyclic values.

3.1 Cyclic Integers

This section shows how adding some rules of modular arithmetic to the specializer formalized in Chapter 2 can unroll loops, make shift offsets static, and eliminate the division and remainder operations inside procedures like `load_sample`.

Figure 3.5 defines the Cyclic domain, redefines M to include Cyclic as a possible meaning, and extends D to handle cyclic values. Whereas previously an integer value was either static or dynamic (either known or unknown), a cyclic value has known base and remainder but unknown quotient. The base must be positive. If the remainder is non-negative and less than the base, then I say the cyclic value is in *normal form*. For now, assume all cyclic values are in normal form.

Figure 3.6 gives one way to extend $\mathcal{S}$ for cyclic values. Again I assume cases not given are avoided by lifting, treating the primitives as unknown (allowing $\diamond$ to match any primitive), or by using the commutivity of the primitives. A case for adding two cyclic values by taking the GCD of the bases is straightforward, but has so far proven unnecessary. Such multiplication is also possible, though more complicated and less useful. The static result of multiplication by zero is a standard online optimization, but is not important to this work.

Figure 3.7 gives rules for `zero?`, division, and remainder. In the case of `zero?`, if the remainder is non-zero modulo the base, then the specializer can statically conclude that the original value is non-zero. But if the remainder is zero, then we need a dynamic test of the quotient. This is a conjunction short-circuiting across stages.

These rules are interesting because the binding times of the results depend on static values rather than just the binding times of the arguments. The result is an example of what in partial evaluation parlance is called *polyvariance in binding times*.

$$
\begin{array}{l}
\mathcal{S}\;\boxed{e_0\texttt{+}e_1}\;\rho = \text{match } (\mathcal{S}\; e_0\; \rho, \mathcal{S}\; e_1\; \rho) \\
\qquad\qquad (\langle b\; q\; r\rangle, s) \rightarrow \text{let } r' = (r+s)\ \%\ b \\
\qquad\qquad\qquad\qquad\qquad\quad q' = (r+s)\ /\ b \\
\qquad\qquad\qquad\qquad\quad \text{in } \langle b\; \boxed{q+q'}\; r'\rangle
\end{array}
$$

$$
\begin{array}{l}
\mathcal{S}\;\boxed{e_0\texttt{*}e_1}\;\rho = \text{match } (\mathcal{S}\; e_0\; \rho, \mathcal{S}\; e_1\; \rho) \\
\qquad\qquad (\langle b\; q\; r\rangle, s) \rightarrow \text{if } s > 0 \text{ then } \langle sb\; q\; sr\rangle \\
\qquad\qquad\qquad\qquad\quad \text{else if } s < 0 \text{ then error} \\
\qquad\qquad\qquad\qquad\quad \text{else } 0
\end{array}
$$

Figure 3.6: Addition and multiplication rules for cyclic values that maintain normal form.

$$
\begin{array}{l}
\mathcal{S}\;\boxed{\texttt{zero?}\quad e}\;\rho = \text{match } \mathcal{S}\; e\; \rho \\
\qquad\qquad\quad \langle b\; q\; r\rangle \rightarrow \text{if } (0 = (r\ \%\ b)) \\
\qquad\qquad\qquad\qquad\quad \text{then let } t = (r\ /\ b) \\
\qquad\qquad\qquad\qquad\qquad \text{in } \boxed{\texttt{zero?}\quad q\texttt{+}t} \\
\qquad\qquad\qquad\qquad\quad \text{else false}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{S}\;\boxed{e_0\texttt{/}e_1}\;\rho = \text{match } (\mathcal{S}\; e_0\; \rho, \mathcal{S}\; e_1\; \rho) \\
\qquad\qquad (\langle b\; q\; r\rangle, s) \rightarrow \text{if } (0 = (b\ \%\ s)) \\
\qquad\qquad\qquad\qquad\quad \text{then } \langle (b\ /\ s)\; q\; (r\ /\ s)\rangle
\end{array}
$$

$$
\begin{array}{l}
\mathcal{S}\;\boxed{e_0\texttt{\%}e_1}\;\rho = \text{match } (\mathcal{S}\; e_0\; \rho, \mathcal{S}\; e_1\; \rho) \\
\qquad\qquad (\langle b\; q\; r\rangle, s) \rightarrow \text{if } b = s \text{ then } r
\end{array}
$$

Figure 3.7: More rules for cyclic values.

The rules for division and remainder could also include cases that returned a dynamic result when the condition on the base is not met. But this would create larger or slower compilers and has not proven to be useful, so my systems just raise an error.

The approach of this chapter has been binding-time improvement by extending the specializer. Instead, one could rewrite the source program to make the separation apparent to an ordinary specializer. This can be done by defining (in the source language) a new type which is just a partially static structure with three members. The rules in Figures 3.6 and 3.7 become procedures operating on this type. The source program must be written to call these procedures instead of the invoking the native arithmetic (this is done with Similix in Appendix A.7).

As it turns out, the rules in Figure 3.6 have several defects. Section 3.3 explains them and a way to overcome them. Note that the solution presented there cannot also be implemented as a manual binding time improvement.

Now I explain the effect of cyclic values on the `sum` example from Figure 3.2. The residual code appears in Figure 3.8. Assumption 1 above directly means that `from` and `to` are cyclic. The end-test for the loop compares these values by calling `zero?` on the difference. As long as the remainders differ, the end-test is statically known to be false. Thus three tests are done in the compiler before it reaches an even word boundary, emits a dynamic test, and forms the loop. All shifts and masks are known at code generation time, allowing immediate-mode instruction selection on common RISC architectures. If one were compiling to VLSI hardware, this might be particularly useful, as these operations become almost free with good layout[1].

These results depend on the style of input. For example the program in Figure 3.9 represents a signal with its start address and length in samples. Since the loop works by counting the samples instead of comparing addresses, no conditionals are eliminated.

Note that the dynamic part of the cyclic values is represented by the quotient. See Section 4.2.5 for more on this.

If the alignments of `from` and `to` had differed, then the odd iterations would have been handled specially before entering the loop. The generation of this prelude code is a natural and automatic result of using cyclic values; normally it is generated by hand or by special-purpose code in a compiler.

---

[1] VLSI represents data with voltages in wires on a two-dimensional surface. With Field Programmable Gate Arrays (FPGA), the hardware can be reconfigured in about a millisecond. [SiHoMcA96] reports on using partial evaluation to dynamically reconfigure FPGA chips.

$\mathcal{S}\ sum$ [$\texttt{from}\mapsto\langle 32\ \boxed{\texttt{fromq}}\ 0\rangle$ $\quad \texttt{to}\mapsto\langle 32\ \boxed{\texttt{toq}}\ 0\rangle$
$\quad\quad \texttt{size}\mapsto 8 \quad \texttt{stride}\mapsto 8 \quad \texttt{r}\mapsto\boxed{\texttt{r}}$]

$\rightarrow$

```
fun sum_0088 fromq toq r =
   if fromq = toq then r else
     sum_0088 (fromq + 1) toq
      (r + (((load_word fromq) >> 0) & 255) +
           (((load_word fromq) >> 8) & 255) +
           (((load_word fromq) >> 16) & 255) +
           (((load_word fromq) >> 24) & 255))
```

Figure 3.8: Assumptions and residual code automatically generated by a specializer with cyclic values.

```
fun sum2 (from, length, size, stride) r =
  if length = 0 then r else
  sum ((from + stride), length-1, size, stride)
      (r + (load_sample from size))
```

Figure 3.9: A slightly different version of `sum` fails to specialize as well.

```
fun vector_signal from to stride size =
 let val from = set_base from 32
     and to = set_base to 32
 in
   fn s_get => load_sample from size
    | s_put => fn v => store_sample from size v
    | s_next => vector_signal (from+stride) to stride size
    | s_end => from = to
  end
```

Figure 3.10: An example use of `set-base`. This code is from the higher-order implementation of the signal interface (Figure 5.4). It is the only use of `set-base` in the implementation.

If we want to apply this optimization to a dynamic value and we can afford to spend some code-space, then we can use case analysis to convert the dynamic value to cyclic before the loop. This results in one prelude for each possible remainder, followed by a single loop, as explained in Section 4.2.3.

Arbitrary arithmetic with pointers can result in values with any base, but once we are in a loop like `sum` we want a particular base. `Set-base` allows the programmer work around this. Figure 3.10 shows an example of its use. Section 4.2.3 explains its implementation.

$$\texttt{(set-base}\ m\ \ b\texttt{)}\ \rightarrow\ \langle b\ \ d\ \ r\rangle$$

While we currently rely on manual placement of `set-base`, we believe automation is possible because the analysis required appears similar to the un/boxing problem [Leroy92].

3.1.1 Multiple Signals

If a loop reads from multiple signals simultaneously, then in order to apply the these optimizations, it must be unrolled until all the signals return to their original alignment. Figure 3.11 illustrates such a situation. The ordinary way of implementing a pair-wise operation on same-length signals uses one conditional in the loop because when one array ends, so does the other. Since our unrolling depends on the conditional, this would result in ignoring the alignments of one of the arrays.

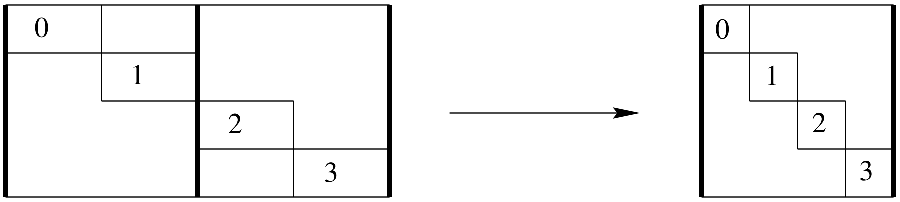

Figure 3.11: Reading a 16-bit signal and writing a 8-bit signal. The input only needs to be unrolled twice, but the output needs to be unrolled four times.

```
fun redbin (from0, to0, size0, stride0)
           (from1, to1, size1, stride1) =
   if ((from0 = to0) andalso (from1 = to1))
   then ()
   else (... ; redbin( ... ))
```

Figure 3.12: Looping over two signals.

To solve this, we perform such operations with what normally would be a redundant conjunction of the end-tests. Figure 3.12 illustrates this kind of loop. In both implementations the residual loop has only one conditional, though after it exits it makes one redundant test[2].

Because 32 has only one prime factor (2), on 32-bit machines this amounts to taking the maximum of all of the signals. If the word-size were composite then more complex cases could occur. For example, a 24-bit machine with signals of stride 8 and 12 results in unrolling 6 times, as illustrated in Figure 3.13.

3.1.2 Irregular Data Layout

The `sum` example shows how signals represented as simple arrays can be handled. The situation is more complex when the data layout depends on dynamic values. Examples of this include sparse matrix representations, run-length encoded vectors (Figure 3.1(c)) , and strings with escape sequences. Figure 3.14 shows how 15-bit values might be encoded into an 8-bit stream while keeping the shift offsets

[2]Simple does this because its compiler to C translates `while(E&&F)S` to `while(E)while(F)S`. Nitrous does this because its input language is in continuation passing style

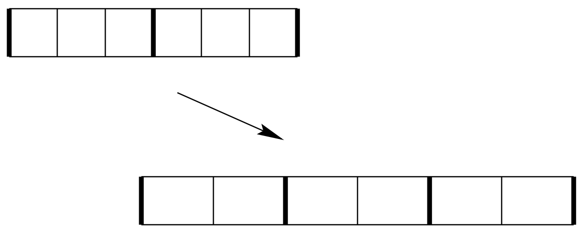

Figure 3.13: Reading a 8-bit signal and writing a 12-bit signal on a machine with 24-bit words.

```
fun read_esc from to r =
    if from = to then r
    else let val v = load_sample from 8
         in if v < 128 then
             read_esc (from + 8) to (v + r)
            else dcall read_esc (from + 16) to
                ((((v & 127) << 8) |
                  (load_sample (from + 8) 8)) + r)
         end
```

Figure 3.14: Reading (and summing) a string of 8-bit characters with escape sequences. Note use of `dcall`.

static. It works because both sides of the conditional of `v` are specialized.

`Read_esc` is a good example of the failure of the dynamic-conditional heuristic. Unless we mark the recursive call as dynamic (so instead of inlining, the memo-table is checked), specialization would diverge because some strings are never aligned, as illustrated in Figure 3.15.

3.2 Sharing and Caching

The remaining inefficiency of the code in Figure 3.8 stems from the repeated loads. The standard approach to eliminating them is to apply common subexpression elimination (CSE) and aliasing analysis (see Section 10.8 of [ASeUl86]) to residual programs. Efficient handling of stores is beyond such traditional techniques, however. We propose fast, optimistic sharing and static caching as an

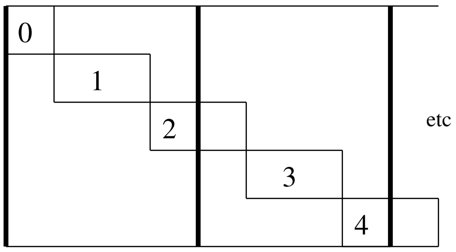

Figure 3.15: A string with escapes illustrating need for `dcall` annotation in `read_esc`.

alternative.

We implement this by replacing the `load_word` primitive with a cached load procedure `load_word_c`. The last several addresses and memory values are stored in a table; when `load_word_c` is called the table is checked. If a matching address is found, the previously loaded value is returned, otherwise memory is referenced, a new table entry is created, and the least recently used table entry is discarded. Part of the implementation appears in Appendix A.1. In fact, any cache strategy could be used as long as it does not depend on the values themselves.

The cache could be held in a global variable if the specializer supports them. We chose to transparently thread an additional argument through all calls and returns by changing its implementation language.

Note that safely eliminating loads in the presence of stores requires negative may-alias information (knowing that values will not be equal) [Deutsch94]. We have not yet implemented anything to guarantee this.

A conspicuous variable is the size of the cache. How many previous loads should be remembered? Though this is currently left to the programmer (with `init-cache` in Nitrous), automation appears feasible. In Nitrous, if the cache is too small then some redundant memory operations will remain; if the cache is too large then excessive and redundant residual code is emitted. In the Simple implementation, there are no dynamic conditionals inside of loops, so this is not an issue.

How does the cache work? Since the addresses are dynamic, the equality test of the addresses used to determine cache membership will be dynamic. Yet these tests must be static to eliminate the cache. Our solution is to use a conservative early equality operator for the cache-hit tests; it appears in Figure 3.16.

This operator takes two dynamic values and returns a static value. The result is

$$
\begin{aligned}
\mathcal{S}\ \boxed{\texttt{early=}\ e_0\ \ e_1}\ \rho = \text{match } (\mathcal{S}\ e_0\ \rho, \mathcal{S}\ e_1\ \rho)\quad & \\
(d_0, d_1) &\to \text{aliases?}(d_0, d_1) \\
(\langle b_0\ q_0\ r_0\rangle, \langle b_1\ q_1\ r_1\rangle) &\to b_0 = b_1 \text{ and} \\
& \quad\ \ \text{aliases?}(q_0, q_1) \text{ and} \\
& \quad\ \ r_0 = r_1
\end{aligned}
$$

Figure 3.16: Rule for `early=`.

true only if the compiler can prove the values will be equal by using positive alias (aka sharing) information. The aliasing information becomes part of the static information given to compilers, stored in the memo tables, etc. For example, a procedure with three dynamic arguments can have five different versions (all equal, none equal, and three ways of two being equal).

In the Nitrous implementation the generated compilers keep track of the names of the dynamic values. The `aliases?` primitive merely tests these names for equality. Thus at compile time a cached load operation requires only a set-membership operation. These names are also used for inlining without a postpass (among other things), so no additional work is required to support `early=`. More explanation appears in Section 4.2.

The cache functions like a CSE routine that examines only loads, so we expect a cache-based compiler to run faster than a CSE-based one. But since CSE subsumes the use of a cache and is probably essential to good performance anyway, why do we consider the cache? Because CSE cannot handle stores, but the cache does, as explained below.

Like the optimizations of the previous section, these load optimizations have been achieved by making the compiler generator more powerful. Even more so than the previous section, the source program had to be written to take advantage of this. Fortunately, with the possible exception of cache size, the modifications can be hidden behind ordinary abstraction barriers.

3.3 Normalization

Now let us see the implication of sharing to the addition rule we defined in Figure 3.6. This addition rule contains a dynamic addition to the quotient. In many cases q' is zero so the addition may be removed by the backend. The Simple system

$$\mathcal{S}\ \boxed{e_0\texttt{+}e_1}\ \rho = \text{match } (\mathcal{S}\ e_0\ \rho, \mathcal{S}\ e_1\ \rho)$$
$$(\langle b\ q\ r\rangle, s) \rightarrow \langle b\ q\ r+s\ \rangle$$

$$\mathcal{S}\ \boxed{e_0\texttt{*}e_1}\ \rho = \text{match } (\mathcal{S}\ e_0\ \rho, \mathcal{S}\ e_1\ \rho)$$
$$(\langle b\ q\ r\rangle, s) \rightarrow \text{if } s > 0 \text{ then } \langle sb\ q\ sr\rangle$$
$$\text{else if } s < 0 \text{ then } \langle\text{-}sb\ \boxed{-q}\ sr\rangle$$
$$\text{else } 0$$

Figure 3.17: Rules for addition and multiplication that depend on late normalization.

```
let val y = (set-base y 4)
    val x = y
    val yy = y + 4
    val xx = x + 4
in (early= xx yy)
end
```

Figure 3.18: With addition rule from Figure 3.6 this would evaluate to false, but with the rule from Figure 3.17, it evaluates to true.

---

works this way. There is a further problem, however: a dynamic addition causes the allocation of a new location, thus losing equality/sharing information (Figure 3.18). The multiplication rule has its own defect: in order to maintain normal form we must dissallow negative scales.

The rules used by Nitrous appear in Figure 3.17. They are simpler and more general because they do not maintain normal form (recall that a cyclic value $\langle b\ q\ r\rangle$ is in normal form if $0 \leq r < b$). Without further adjustment, use of the new addition rule would result in frequent non-termination because there is no case that emits dynamic additions. To compensate, cyclic values are returned to normal form at memo points: I call this *late normalization* of cyclic values. The extra information propagated by this technique (early-equality through fixed points) is required to handle multiple overlapping streams of data.

Nitrous implements this as follows. When a compiler begins to make a dynamic code block, all cyclic values are normalized by adjusting *r* and emitting counter-acting additions to q. The sharing between these values must be maintained across this adjustment. Identifying all cyclic values is difficult because they may be hidden in the stack or closures, see Section 4.2.5.

3.4 Store Caching

So far we have only considered reading from memory, not writing to it. Storing samples is more complicated than loading for two reasons: an isolated store requires a load as well as a store, and optimizing stores most naturally requires information to move backwards in time. This is because if we read several words from the same location, then the reads after the first are redundant. But if we store several words to the same location, all writes before the last write are redundant.

We can implement `store_word_c` the same way a hardware write-back cache does ([HePa90] page 379 in the second edition): cache lines are extended with a dirty flag; stores only go to memory when a cache line is discarded. The time problem above is solved by buffering the writes.

The load is unnecessary if subsequent stores eventually overwrite the entire word. I achieved this by extending the functionality of the cache to include not just dirty lines, but partially dirty lines (this is sometimes called *sectoring*) Thus the status of a line may be either clean or a mask indicating which bits are dirty and which are not present in the cache at all. When a clean line is flushed no action is required. If the line is dirty and the mask is zero, then the word is simply stored. Otherwise a word is fetched from memory, bit-anded with the mask, bit-ored with the line contents, and written to memory. Note that the masks in the cache lines imply that the implementation substrate support native-sized integers.

Here is an example. Figure 3.19 shows code that stores an increasing series of integers into every sample of a signal. Figure 3.20 shows the result of specializing it to a case where loads are unnecessary. Figure 3.21 shows the result when the stores are non-continuous, resulting in a different expansion of the `flush_line` procedure (see Appendix A.1). In these figures, the delaying effect of the cache on writes has been removed to make the code clearer. Actual residual code from Nitrous for the continuous case appears in Figure 3.22. Note how the dynamic part of cache appears as arguments to procedures, and the duplication of the procedure into entry and steady-state (write-pending) versions.

```
fun ramp (from, to, size, stride, i) =
  if from = to then () else
  (store_sample(from, size, i);
   ramp (from+stride, to, size, stride, i+1))
```

Figure 3.19: Fill a signal with a ramp (sequential integers).

```
fun ramp_0088 (from, to, i) =
  if from = to then () else
  let i1 = i+1     i2 = i1+1
      i3 = i2+1    i4 = i3+1
  (store_word(from, i | i1<<8 | i2<<16 | i3<<24);
   ramp_0088 (from+1, to, i4))
```

Figure 3.20: Specialized to continuous writes avoids loads.

```
fun ramp_00816 (from, to, i) =
  if from = to then () else
  let i1 = i+1     i2 = i1+1
  (store_word(from, ((load_word from) & 0xff00ff00)
                    | i | i1<<16);
   ramp_00816 (from+1, to, i2))
```

Figure 3.21: Specialized to signal with gaps, masked loads appear.

```
ramp_0088(k from to i) {
  if (from = to) (car k)()
  let i1 = i+1      i2 = i1+1
      i3 = i2+1     i4 = i3+1
  ramp2(k from+1 to i4 from
        (i<<0 | i1<<8 | i2<<16 | i3<<24))
}

ramp2(k from to i a v) {
  if (from = to) (store_word(a, v);
                  (car k)())
  let i1 = i+1      i2 = i1+1
      i3 = i2+1     i4 = i3+1
  store_word(a, v);
  ramp2(k from+1 to i4 from
        (i<<0 | i1<<8 | i2<<16 | i3<<24))
}
```

Figure 3.22: Actual residual code from Nitrous in `root` showing threading of dynamic part of cache. The cache is held in the variables `a` and `v`. The notation is the intermediate language described in Section 4.1.

3.5 Correctness

This section discusses (but does not prove) the correctness of $\mathcal{S}$ and of bit-addressing. In the introduction, I claimed that specialization "preserves the semantics" of programs. Thus the standard measure of correctness of specializers is satisfaction of the first Futamura projection:

$[\![f]\!]\ x\ y =_\alpha [\![\ [\![spec]\!]\ f\ x]\!]\ y$

Note that this is a strong notion of correctness, defined by equivalence of terms. [GoJo91] contains such a proof for λ-mix where the equivalence permits α-conversion (renaming variables). A similar proof for $\mathcal{S}$ would be unremarkable, except for the handling of the extension for cyclic integers. Here, the proof would use algebraic properites of the rules in Figures 3.6 and 3.7. Because $\mathcal{S}$'s rules for division and remainder may report errors, and because $\mathcal{S}$ may not terminate, we can only prove *partial* equivalence, that is, if both sides of the equation are defined, then they are equivalent (up to α-conversion). Note that the correctness of polyvariant specialization and memoization has so far remained unaddressed.

Safe handling of side-effects requires maintaining the order of computations. I believe the standard solution of let-insertion [BoDa91] could be incorporated to $\mathcal{S}$ without problems. Nitrous preserves order of computation because it processes programs in continuation-passing style.

However, treating memory operations as generic side-effects is too restrictive (an exception is when the memory is mapped to an I/O device). Showing that bit-addressing is correct requires proving that the cache preserves the ordinary semantics of a memory system[3]. Proofs of this kind are standard and, with one exception, I see no problem developing one for a software cache.

The problem stems from the conservative early equality. Correctness depends either on finding some way to guarantee negative may-alias information, or a relaxed definition of correctness. One way to make the guarantee is to dynamically verify pointer inequality (for example, before entering a loop). If the data layout is irregular then maintaining the guarantee is very difficult. Until an inexpensive way to make guarantees is found, I choose to live dangerously.

Correctness of sharing is part of the satisfaction of the Futamura projection given above.

Late normalization is more interesting. Showing that the modified rules of Figure 3.17 are algebraically correct is not hard. The danger of late normalization

[3]The ordinary semantics of memory guarantee that reading location p returns the last value written to location p.

```
f : a -> c
g : c -> b
l : a list
map : (a -> b) -> a list -> b list

map f (map g l) → map (f o g) l
```

Figure 3.23: Deforestation transforms a two-pass procedure to a one-pass procedure. Infix composition of functions is denoted with `o`, as usual.

is non-termination, so a proof of partial equality is not revealing. The useful proof is that late normalization does not introduce non-termination.

3.6 Limitations

Specialization, as described in Chapter 2 and extended in Chapter 3, is unable to perform many useful optimizations. This section give some examples and suggests how to achieve them.

An important transformation on programs that process streams of data is loop fusion. In the functional language community, a general form of this optimization is known as *deforestation* [Wadler88]. This transformation eliminates intermediate data structures, as illustrated in Figure 3.23. Furthermore, optimizing compilers for numerical and data-parallel languages such as High Performance Fortran and NESL perform extensive analysis to determine how to divide the computation into passes, layout the data-structures in memory, and coordinate multiple processors [SSOG93, ChaBleFi91].

Partial evaluation alone does not solve these problems. Specializing `map f (map g l)` has no effect. Instead, my approach is to specify procedures directly in one pass, and use specialization to efficiently implement `(f o g)`. Similarly, I believe specialization could be effective as a middle stage in such a compiler.

Another important kind of code one would like to remove is clipping and array-bounds checks. For example, frequently one can guarantee that some raster operations will be unobscured and in-bounds, and thus avoid clipping operations. The natural way to handle this with a specializer is to propagate additional static information. For example, one might statically know that `10 < i < 100`. As usual, this can be done by manual binding-time improvement or by improving the

```
fun bitcopy (start, stop, start0) =
  if start=stop then ()
  (store_sample(start0, 1, load_sample(start, 1));
   bitcopy(start+1, stop, start0+1))
```

Figure 3.24: Bit-serial copy.

```
fun bitcopy_000(start, stop, start0) =
  if start=stop then ()
  let v = load_word(start)
  store_word(start0, (v&1) | (((v>>1)&1)<<1)
                     (((v>>2)&1)<<2) | (((v>>3)&1)<<3) |
                     (((v>>4)&1)<<4) | (((v>>5)&1)<<5) |
                     (((v>>6)&1)<<6) | (((v>>7)&1)<<7));
   bitcopy_000(start+1, stop, start0+1)
```

Figure 3.25: Residual code with 8-bit word-size showing redundant un/marshall.

specializer. The latter route leads to the technique of *generalized partial computation* [FuNoTa91, SoGluJo96], wherein PE is extended with a theorem prover.

Say we wrote bitcopy in bit-serial fashion, as in Figure 3.24. The rules of this chapter produce (assuming an 8-bit word for brevity) the obviously inefficient code in Figure 3.25.

The problem is that the source program uses a single bit of the input stream as an intermediate value. The bits are then just reassembled into words. There appear to be two parts to the problem: tracking where individual bits of data go, and then when a word is finally really needed, determining how to assemble that bit-pattern with available hardware (in the above case, it would determine that no work at all is needed).

We now speculate on how to provide this optimization with a specializer. Introduce a new binding time, that is another kind of partially static integer: masked. The static part includes a mask indicating which bits are known, what those bits are, and source information for each dynamic bit. Operations such as shifting and masking can then be performed statically, simply by rearranging the dynamic bits. When one of these values is passed to an unknown primitive, it is lifted to a simple value, and the assembly routine is invoked.

3.7 Summary

This chapter added cyclic integers to the specializer from the last chapter. Use of these partially static integers combined with the memoization results in smart loop unrolling. In order to optimize memory operations with stores, we implement a cache in software. The cache is controlled by aliasing information in the compiler. Together, these allow us to write signal processors independent of the signal's representation (at least, relativly independent when compared to other languages). The results of building some software with this kind of interface appears in Chapter 5. The next chapter (Chapter 4) explains how the Nitrous implementation works in detail.

Chapter 4

Nitrous

Nitrous[1] is a compiler generation system. It accepts a program in a full-featured source language and transforms the program into a compiler. Because it runs very slowly and the constant propagation is aggressive, excessively large compilers and residual code is a problem. The objective of the design was to demonstrate that enough static information is available to make transformations like those of Section 1.5 and Chapter 3 work. For example, one might wonder if higher-order functions or safety conflicts with late normalization of partially static integers or early equality.

`Root` is the intermediate language at the center of the system, it is described below. Figure 4.1 identifies the three kinds of program transformations within Nitrous:

- Front-ends which produce `root` programs from programs in user-defined languages. Though the front-ends discussed here are all automatically generated by the system, the design is meant to support hand-written front-ends, such as an ordinary ML compiler.

- A specializer for an untyped intermediate language `root`.

- A back-end provides machine-code assembly, optimization, and the rest of the run-time system.

[1]It is named for nitrous oxide N_2O, a gas used to make cars go really fast. It is also a common medical anaesthetic. William James found that its subjective effect was to reveal the Hegelian synthesis [James1882].

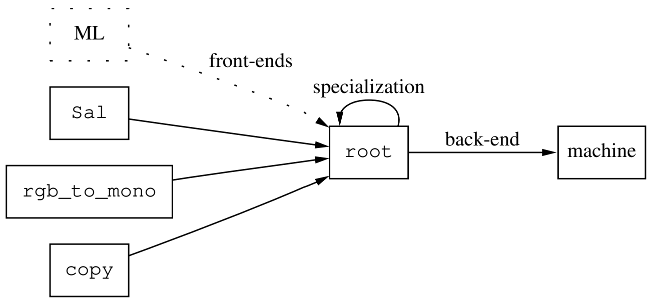

Figure 4.1: Diagram showing components of the Nitrous system. Boxes denote languages, and arrows denote program transformations. The ML front-end is hypothetical, and so is drawn with dotted lines.

The specializer is a compiler generator `cogen`. This chapter explains the mechanics of how `cogen` works, including the intermediate language and some important front-ends.

`Root` is a simple abstract machine code, like higher-order three-address-code ([ASeUl86], p. 466) with an unlimited number of registers and in continuation-passing closure-passing style (CPS-CPS) [Appel92]. Thus the stack in a `root` program is explicitly represented as a data structure. The model includes data structures, arithmetic, an open set of primitive functions, and represents higher-order values with closures.

A language supports *reflection* if it can convert a data structure representing an arbitrary program text into a value (possibly a function). Lisp supports reflection with `eval` and `compile`. *Reification* is the opposite of reflection, that is, converting a value (possibly a function) into a corresponding text. Although reification is commonly available in debuggers and interpreters, not even Common LISP standardizes it. [FriWa84] defines and discusses these ideas in detail. Reification is sometimes called introspection [Thiemann96] Section 4.

`Root` supports both reflection and reification. By supporting reflection we make code-producing functions first class. Nitrous takes this a step further by using reification to make the compiler-producing function first class: rather than working with files, cogen just maps procedures to procedures.

Sal is a recursive-equations language augmented with `lambda` and various convenience features. The `cogen`-created compiler produces straight-forward code. It performs CPS-CPS conversion and compiles tail-recursive calls without building stack frames, but is otherwise non-optimizing. Sal is covered in Section

4.3.

`Rgb_to_mono` (from Figure 1.4) and `copy` (from Figure 5.4) are two example media processing languages. The interpreters for these languages were written in Sal.

Nitrous implements specialization in two stages: `cogen` transforms a `root` program and *binding times* (BTs) for its arguments into a *generating extension*. The extension is executed with the static values to produce the specialized residual program. The BTs categorize each argument as static program or dynamic data. The extension consists of a memo table, followed by the static parts of the computation interleaved with instructions that generate residual code (i.e. do RTCG).

Because the compilers produce the same language that `cogen` accepts, and the `root` text of the residual programs is easily accessible, multiple layers of interpretation can be removed by multiply applying `cogen`. For example the output of the Sal compiler is valid input to `cogen`. This is how `rgb_to_mono` and `copy` were built. The lift compiler (see Section 4.2.3) also uses two layers. Another possibility is to include a compiler generator as a primitive in Sal.

Multi-stage application requires that the generated compilers create correctly annotated programs, which can be difficult. In [GluJo94, GluJo95] Glück and Jørgensen present more rigourous and automatic treatments of layered systems using specializer projections and multi-stage binding-time analysis.

This chapter is organized as follows. First, the intermediate language is presented, followed by the compiler generator, the Sal front end, and the conclusion. The bulk of the material consists of technical details of the compiler generator.

4.1 The Intermediate Language

The core of the system is the intermediate language. Its formal syntax appears in Figure 4.2. A program is called a code pointer, or just a `code`. When a `code` is invoked, its formal parameter list is bound to the actual arguments. The list of `prim` and `const` instructions execute sequentially, each binding a new variable. `If` tests a variable and follows one of two instruction lists. These lists terminate with a `jump` or `exit` instruction. A `jump` invokes the `code` bound to the first argument. The remaining arguments form the parameter list, and the cycle repeats itself. `Exit` stops the machine and returns an answer. Formal semantics appear in Figure 4.3. The semantics are given in Scheme [R4RS], extended with a pattern-matching macro.

code ::= `(code` *name* *args* *instrs*`)`
instr ::= `(prim` *v* *prim* `.` *args*`)`
  | `(const` *v* *constant*`)`
  | `(if` *v* *true-branch*`)`
  | `(jump` *v* `.` *args*`)`
  | `(exit` *v*`)`
v ::= *variable*
instrs ::= *instr*$^{+}$
args ::= *variable**
true-branch ::= *instrs*
prim ::= *primitive operation*

Figure 4.2: `Root` syntax. As usual, * and + superscripts denote lists of length zero-or-more and one-or-more, respectively.

```
(define (eval-root code args)
  (match code
    (('code name formal-args source-code)
     (let loop ((instrs source-code)
                (env (zip formal-args args)))
       (let ((lu (lookup env)))
         (match instrs ; not just the car
           ((('exit arg)) (lu arg))
           ((('jump fn . args))
            (eval-root (lu fn) (map lu args)))
           ((('if pred t-branch) . f-branch)
            (if (lu pred)
                (loop t-branch env)
                (loop f-branch env)))
           ((('const var c) . rest)
            (loop rest (cons (cons var c) env)))
           ((('prim var prim . args) . rest)
            (let ((v (apply prim (map lu args))))
              (loop rest (cons (cons var v) env))))))))))
```

Figure 4.3: Semantics of `root` in Scheme.

```
(code append (k l m)
  ((prim t0 null? l)
   (if t0 ((prim t1 car k)
           (jump t1 k m)))     ; a
   (prim frame list k l m)
   (const cont cont)           ; b
   (prim cl close cont frame) ; c
   (prim t2 cdr l)
   (const append append)       ; b
   (jump append cl t2 m)))

(code cont (self r)
  ((prim t0 cdr self)
   (prim k car self)
   (prim t1 cdr t0)
   (prim l car t1)
   (prim t2 cdr t1)
   (prim m car t2)
   (prim t3 car l)
   (prim nr cons t3 r)
   (prim t4 car k)
   (jump k nr)))
```

Figure 4.4: `Root` code for `append`, in literal syntax. See text for notes.

Structured higher-order control flow is managed with closure-passing [Appel92]. A closure is a pair consisting of a code pointer with its bound variables, and is invoked by jumping to its left member and passing itself as the first argument. A normal procedure call passes the stack as the next argument. The stack is just the continuation, which is represented with a closure.

See Figure 4.4 for an example program in `root` that uses a stack. Notes: *a* return by jumping to the car of `k`, passing `k` and `m` as arguments. *b* These constants are codes not symbols (`append` is a circular reference). *c* `close` is like `cons`, but identifies a closure. Figure 4.5 shows the same program in the abbreviated syntax used in the remaining examples. *d* destructuring assignment expands into car/cdr chain. Figure 4.7 shows how to evaluate this program. Figure 4.6 shows a higher-order program.

```
append(k l m) {
  if (null? l) (car k)(k m)      ; a
  frame = (list k l m)
  cl = (close cont frame)        ; b c
  append(cl (cdr l) m)           ; b
}
cont(self r) {
  (k l m) = (cdr self)           ; d
  nr = (cons (car l) r)
  (car k)(k nr)
}
```

Figure 4.5: `Root` code for `append`, in sugary syntax. See text for notes.

```
compose(k f g) {
  lam = (close apply_g (list f g))
  (car k)(k lam)
}
apply_g(k self x) {
  (f g) = (cdr self)
  cl = (close cont (list k f g))
  (car g)(g cl x)
}
cont(self r) {
  (k f g) = (cdr self)
  (car f)(f k r)
}
```

Figure 4.6: `Root` code for `compose`, showing how closures and higher-order calls work.

```
(define top-cont '((code top-cont (self r) ((exit r)))))

(eval-root append-root (list top-cont '(1 2 3) '(4 5)))

→

(1 2 3 4 5)
```

Figure 4.7: Session with Nitrous showing execution of the program `append` (which is bound to the Scheme variable `append-root` to avoid conflict with the native version). `Top-cont` is a stub continuation that returns a value to the Scheme metalevel.

Factors that weigh in favor of an intermediate language like `root` instead of a user-level language like Scheme or SML: it makes `cogen` smaller and easier to write; it provides a target for a range of source languages; it provides an interface for portability; it exposes language mechanism (such as complex optional arguments, pattern matching, method lookup, or multi-dimensional arrays) to partial evaluation; it reduces assembly overhead because it is essentially an abstract RISC code.

Two factors that weigh against `root`: explicit types would simplify the implementation and formalization, and on some architectures good loops (e.g. PC-relative addressing or other special instructions) are difficult to produce. Using a language like the JVM [GoJoSte96], or one of the intermediate languages of [TMCSHL96] would leverage existing research.

4.2 The Compiler Generator

`Cogen` is directly implemented (rather than produced by self-application), polyvariant in binding times (it allows multiple binding time patterns per source procedure), performs polyvariant specialization with memo-tables, and handles higher-order control flow. This section summarizes how `cogen` and its extensions work. The subsections cover the implementation of binding times, cyclic integers, lifting, static extensions, special primitive functions, shape un/lifting, and dynamic control stacks.

While Nitrous does very well with small examples, several flaws make scaling difficult. Nitrous is apt to classify too much computation as static and produce

```
(define append-DSD (cogen append-root
                          '(dynamic static dynamic)))

(eval-root append-DSD
          `(,top-cont
            (1 2 3)
            ,(var->shape 'k)
            ,(var->shape 'm)))
```

$\rightarrow$

```
append-123(k m) {
  (car k)(k (cons 1 (cons 2 (cons 3 m))))
}
```

Figure 4.8: Session with Nitrous showing generation and execution of an extension.

large amounts of residual code, thus requiring the programmer to make careful use of lifting. Several expediencies made its developement easier, but increased the code output. Section 5.3 describes an implementation that scales better by using a restricted language.

One interesting feature of Nitrous is that the input language is in continuation-passing style. This has three results: let-insertion for safety is avoided, dynamic choice of static values is supported, and the generated compilers do CPS transformation.

Although the input language is not explicitly typed, it turns out that annotations marking particular values as stacks and closures are essential. This indicates that an explicitly typed input language would work better.

`Cogen` converts a `code` and a binding-time pattern to a *metastatic extension*. A metastatic extension is denoted with the name of the code pointer and the BT pattern, for example `append(D S D)`. The extension takes the static values and the names of the dynamic values and returns a procedure—the specialized code. Figure 4.8 shows an example session.

Inlining is controlled by the dynamic conditional heuristic (see Section 2.2), but setting the special `$inline` variable overrides the heuristic at the next jump (see Figures 4.18 and 4.22 for examples).

In CPS-CPS continuations appear as arguments, so static contexts are natu-

```
dyn-if(k s d) {
  frame = (list k s d)
  cl = (close cont frame)
  if (d) (car cl)(cl 2)
  (car cl)(cl 3)
}
cont(self r) {
  (k s d) = (cdr self)
  rr = r + s
  (car k)(rr)
}
```

```
dyn-if2(k d) {
  if (d) cont1(k)
  cont2(k)
}
cont1(k) {
  (car k)(k 2)
}
cont2(k) {
  (car k)(k 3)
}
```

Figure 4.9: Propagating a static context past a dynamic conditional. The source code is on the left, and the residual code on the right.

rally propagated. Figure 4.9 shows the translation of `(+ s (if d 2 3))` into `root` and the result of specialization. This is the effect refered to in Section 2.1.

4.2.1 Binding Times

Binding times are properties derived from the interpreter text while a compiler generator runs. In the formal system of Chapter 2, they were the injection tags on values in M. Primarily they indicate if a value will be known at compile time or at run-time, but they are often combined with the results of type inference, control flow analysis, or other static analyses. Binding times are sometimes called *metastatic* values because in self-application of a specializer, they are static at the meta-level (the outer application).

`Cogen`'s binding times are described by the grammar in Figure 4.10. The binding times form a lattice where $\mathtt{D} \sqsubset \mathtt{C} \sqsubset \mathtt{S} \sqsubset (\widetilde{const}\ c)$ and $\mathtt{D} \sqsubset (\widetilde{cons}\ bt\ bt) \sqsubset \mathtt{S}$. Thus `D` is the bottom of the lattice, though this contradicts the metaphore suggested by “lifting”. Circular binding-times are allowed, so binding-times may have infinite size as long as they are recognized by a context-free grammar. The join operation of the lattice is not used by Nitrous, but the following equations define it for finite binding-times:

$x \sqcup \mathtt{D} = \mathtt{D}$
$(\widetilde{cons}\ x_0\ y_0) \sqcup (\widetilde{cons}\ x_1\ y_1) = (\widetilde{cons}\ (x_0 \sqcup x_1)\ (y_0 \sqcup y_1))$
$(\widetilde{const}\ c_0) \sqcup (\widetilde{const}\ c_1) = \mathtt{S}$ *unless* $c_0 = c_1$

$$\begin{aligned} bt ::= {} & \texttt{D} \mid \texttt{S} \mid \texttt{C} \\ & \mid (\widetilde{const}\ \ c) \\ & \mid (\widetilde{cons}\ \ bt\ \ bt) \end{aligned}$$

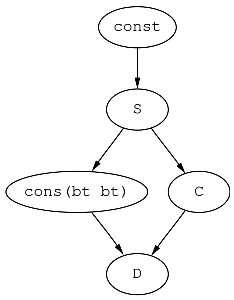

Figure 4.10: The binding time lattice. In the grammar on the left `D` denotes dynamic, `S` denotes static, `C` denotes cyclic, and c denotes a constant. The lattice order is illustrated on the right.

$\texttt{C} \sqcup (\widetilde{cons}\ x_0\ y_0) = \texttt{D}$

Cons cells are handled by representing binding times with graph-grammars as Mogensen did. Pairs in binding times are labeled with a *cons point* that identifies which instruction the pair came from. If the same label appears on a pair and a descendant of that pair then the graph-grammar is collapsed, perhaps forming a circularity [Mogensen89]. Figures 4.11 and 4.12 show two possibilities.

The function of the cons-points would be better provided by explicit inductive types.

We denote such a pair $(\widetilde{cons}\ \ bt\ \ bt)$ (the label is invisible here). $(\widetilde{list}\ \ x\ \ y\ \ \texttt{...})$ abbreviates $(\widetilde{cons}\ \ x\ \ (\widetilde{cons}\ \ y\ \ \texttt{...}\ \ \ (\widetilde{const}\ \ \texttt{nil})))$. We use familiar type constructors to denote circular binding times. Figure 4.13 depicts several useful examples.

As in Schism [Consel90], control-flow information appears in the binding times. `Cogen` supports arbitrary values in the binding times, including `code` pointers, the empty list, and other type tags. Such a binding time is denoted $(\widetilde{const}\ \ c)$, or just *c*.

Closures are differentiated from ordinary pairs in the `root` text, and this distinction is maintained with a bit in $\widetilde{cons}$ binding times. Such a binding time is denoted $(\widetilde{close}\ \ bt\ \ bt)$.

An additional bit on pair binding times supports a sum-type with atoms. In terms of a grammar, a $\widetilde{cons}$ with this bit set may be either a pair or an atom (a terminal). The bit is set during collapse if a cons is summed with an atom; in Figure 4.11 the bit is set on `alpha`. In Figure 4.12, the bit is not set. The bit is

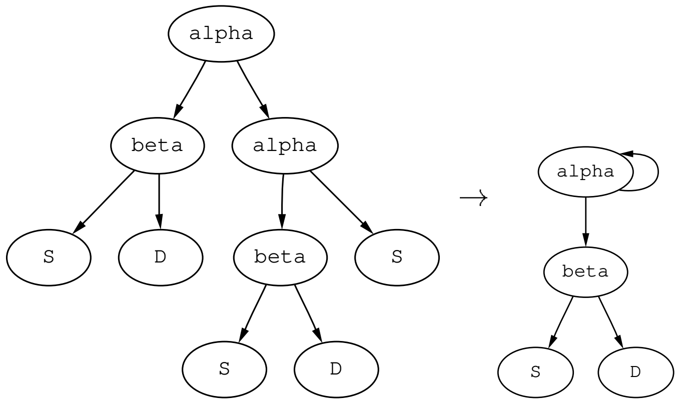

Figure 4.11: The Cons-point `alpha` appears twice and the graph is collapsed, forming a circularity.

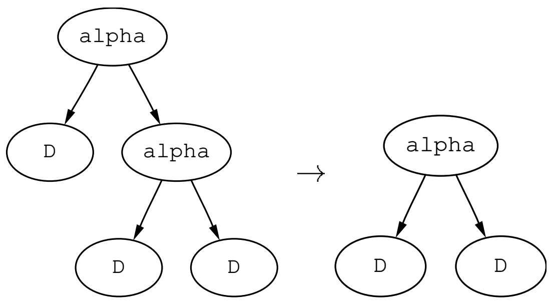

Figure 4.12: The Cons-point `alpha` appears twice and the graph is collapsed, not forming a circularity.

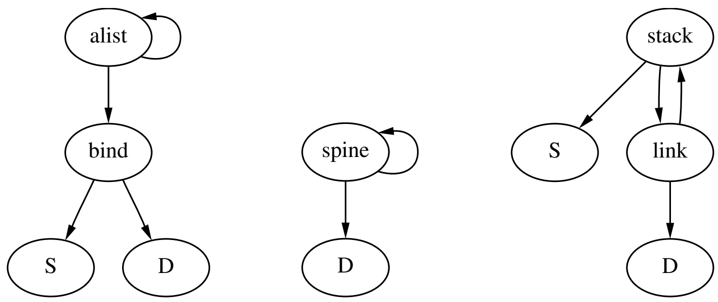

Figure 4.13: Three binding times: `(S * D) list` is typically used as an association list from static keys to dynamic values, `D list` is a list only whose length is static, *stack-bt* is the binding time of a control stack.

not denoted.

Nitrous can break an integer into static base, dynamic quotient, and static remainder, as described in Section 3.1. In the lattice, $\mathtt{S}\sqsubset\mathtt{C}\sqsubset\mathtt{D}$. We have to make special cases of all the primitives that handle cyclic values; these were introduced in Section 3.1. See Section 4.2.4 for a description of the other special primitives.

4.2.2 Shapes and Sharing

Extensions rename the variables in the residual code and keep track of the *shapes* (names and cons structure) of the dynamic values. The shape of a simple dynamic value is the name of the variable that will hold the value. If the shape of a value is a pair of names, then the dynamic value is held in two variables. Shapes always mirror the binding times. In partial evaluation jargon, the effect of using shapes is called variable splitting or arity raising.

In Figure 4.8, the binding time of the third argument is `D`, so its shape must be just a name. In Figure 4.14, the binding time is `D list`, so the shape is a list of names. In Figure 4.15 the binding time is the same, but because Nitrous uses the *sharing* information from the identical names in the shape, only two values are passed instead of four.

Shapes are part of the key in the static memo-table. Two shapes match only if they have the same aliasing pattern, that is, not only do the structures have to be the same, but the sharing between parts of the structures must be the same. This can be computed in nearly-linear time.

The effect of variable splitting is that the members of a structure can be kept

```
(define append-DSD*
  (cogen append-root
         `(dynamic static ,(make-list-bt 'dynamic))))

(eval-root append-DSD*
           `(,top-cont
             (2 3)
             ,(var->shape 'k)
             ,(map var->shape '(a b c))))
→
append-23abc(k a b c) {
  t = (cons a (cons b (cons c '())))
  (car k)(k (cons 2 (cons 3 t)))
}
```

Figure 4.14: Session with Nitrous showing the effect of shapes. `Make-list-bt` creates `D list` from `D`.

in several registers, instead of allocated on the heap (abstracted into one register). The effect of sharing is that if two of the members are known to be the same, they only occupy one register. This comes down to adding equational constraints to the static information. This is why the residual code from `rgb_to_mono` in Figure 1.1 has fewer arguments than Figure 1.2.

4.2.3 Lifting

Lifting is generalization, or abstracting away information. If we abstract away the right information the compiler will find a match in its memo table, thus proving an inductive theorem and forming a dynamic loop. The simplest lift converts a known value to an unknown value in a known location (virtual machine register). Lifting occurs when

- a metastatic `code` is converted to a static value.
- a `prim` has arguments of mixed binding time, causing all the arguments to be lifted.
- a `jump` has dynamic target, causing all the arguments to be lifted.

```
(eval-root append-DSD*
           `(,top-cont
             (2 3)
             ,(var->shape 'k)
             ,(map var->shape '(a b b a))))
→
append-23abba(k a b) {
  t = (cons a (cons b (cons b (cons a '()))))
  (car k)(k (cons 2 (cons 3 t)))
}
```

Figure 4.15: Session with Nitrous showing the effect of sharing.

- a Cons-point repeats in a binding-time grammar, causing its collapse (Figures 4.11 and 4.12).
- a lift directive appears, as a result of manual annotation, or produced by a delayed lift.

Lifting is inductively defined on the binding times. The base cases are:

1. `S` $\rightarrow$ `D` allocates a dynamic location and initializes it to the static value.
2. `C` $\rightarrow$ `D` emits a multiplication by the base (unless it is one) and addition with the remainder (unless it is zero).
3. `S` $\rightarrow$ `C` results from an annotation used to introduce a cyclic value. The conversion is underconstrained; currently a base of one is assumed. This has been obviated by `set-base`.
4. `(`$\widetilde{cons}$ `D D)` $\rightarrow$ `D` emits a dynamic `cons` instruction.
5. `(`$\widetilde{close}$ `(`$\widetilde{const}$ p`)` *frame*`)` $\rightarrow$ `D` generates and inserts a call to p `( (`$\widetilde{close}$ `D` x`) D D ...)` (all but the first argument are `D`), then emits the cons pairing the specialized code with the dynamic values.
6. `(`$\widetilde{close}$ `S` *frame*`)` $\rightarrow$ `D` This is lifting a static extension to dynamic, so it is like the previous case, but the binding times are passed to the extension.

Cases 5 and 6 are particularly interesting. Any static information in *frame* is saved by *reassociating* it into the code pointer before it is lifted. This introduces a complication, as explained in Section 4.2.5 below.

Manual lifting is supported in `root` with an instruction understood by `cogen` but ignored by the `root` semantics:

instruction ::= ...
  | `(lift` *v*`)`
  | `(lift` *v* *bt*`)`
  | `(lift` *v* `(`*args*`)` *proc*`)`

The variable *v* is lifted to `D`, unless the target *bt* is given. Any legal lift is supported, including lifting to/from partially static structures with loops and closures. Instead of giving a binding time *bt*, one can give a procedure *proc* which is executed on the binding times of *args*. This provides a gateway to Scheme for the lift language.

A *delayed lift*, denoted $\mathtt{lift}_t$, produces a residual lift at time $t - 1$. If t is zero, then it functions as an ordinary lift defined above.

Lifting Structures

If a lift is not one of the base cases outlined above, then the *lift compiler* is invoked to create a procedure that takes the value apart, applies simple lifts at the leaves, and reassembles the structure. `Cogen` inserts a call to this procedure into the source program, and recurses into it.

For example, consider the lift `(S * D) list`$\rightarrow$`D`. The compiler has a list of values and a list of variable names. It recurses down the lists, and emits a const and a cons instruction (making a binding) for each list member. At the base it recovers the terminator, then it returns up and emits cons instructions that build the spine. The copy function appears in Figure 4.16, and its specialization in Figure 4.17.

It turns out that this lift compiler can be created by `cogen` itself. The meta-interpreter is just a structure-copy procedure that traverses the value and the binding times in parallel. A *delayed lift* annotation is used where the BTs indicate a simple lift. Specializing this to the binding times results in a copy function with lifts at the leaves. The value passed to the copy function *has* the binding time that was just a static value. When the continuation is finally called the remaining static information is propagated. The copy function may contain calls to itself (where the BT was circular) or to other extensions (to handle higher-order values).

```
(define (copy_sdl2d l)
  (if (null? l)
      '()
      (let ((hd (car l))
            (tl (cdr l)))
        (cons (cons (lift (car hd))
                    (cdr hd))
              (copy_sdl2d tl)))))
```

Figure 4.16: Copy-function with lifts at the leaves. It is generated by the lift compiler to perform a lift from `(S * D) list` to `D`.

```
sdl2d1(k a b) {
  (car k)(k (cons (cons a 1) (cons (cons b 2) '())))
}
```

Figure 4.17: Result of specializing copy function.

This is an example of multi-stage compiler generation because the output of a generated compiler is being fed into `cogen`. The implementation requires care as `cogen` is being used to implement itself, but the possibility of the technique is encouraging.

Controlling Cyclic Values

`Set-base` is an special annotation that takes an integer with any binding time and converts it to cyclic with a particular base. This can be a lift, for example when going from static to cyclic base four, or when going from cyclic base four to base two. Note that cyclic base one is functionally the same as dynamic. Converting from dynamic to cyclic requires case-analysis (known as The Trick in partial evaluation jargon). Changing the base of a cyclic value may require both of these, for example to go from base two to base three.

Lifting out a factor is easy because static information is just discarded from the compiler. Case-analysis creates static information by emitting dynamic conditionals and duplicating code. Nitrous implements this by inserting a call to the procedure in Figure 4.18. The argument `d` is the dynamic value to be converted; `n` is its maximum possible value plus one; `n` is static. Note that the test for the error

```
finite-dynamic-to-static(k d n) {
  if (zero? n) error
  n1 = n - 1
  if (n1 = d) $inline = #t
              (car k)(k n1)
  $inline = #t
  finite-dynamic-to-static(k d n1)
}
```

Figure 4.18: Convert a dynamic value to static by case analysis.

is essential even though it may never be true, as without it the compiler will not terminate.

4.2.4 Details and Complications

Special Primitive Functions

`Cogen` treats some primitive functions specially, generally in order to preserve partially static information. Figure 4.19 gives the improved binding times possible, and in what situations they occur. Notes:

a implement copy propagation by just copying the shape.

b because `root` is untyped.

c for variable splitting.

d the result is metastatic if the pair has never been joined with an atom (see Section 4.2.1). `Null?` and `atom?` are also supported.

e `apply` takes two arguments: a primitive (in one back-end a C function, in the other a Scheme procedure) and a list of arguments. If the primitive and the number of arguments are static, then the compiler can just generate the primitive instead of building an argument list and generating an `apply`. This supports interpreters with an open set of primitives or a foreign function interface. Notice this does not improve the binding times, it just generates better code.

`(identity` x`)`	x	a
`(cons S S)`	`S`	b
`(cons S D)`	`(`$\widetilde{cons}$ `S D)`	
`(cons D D)`	`(`$\widetilde{cons}$ `D D)`	c
`(car (`$\widetilde{cons}$ x y`))`	x	
`(pair? (`$\widetilde{cons}$ `_ _))`	`S`	d
`(apply S (D list))`	`D`	e
`(+ C S)`	`C`	
`(* C S)`	`C`	
`(+ D S)`	`C`	
`(* D S)`	`C`	
`(zero? C)`	`S D`	f
`(imod C S)`	`S`	
`(idiv C S)`	`C`	
`(early= D D)`	`S`	g

Figure 4.19: See the text for notes.

f extensions for both `S` and `D` are created, the compiler chooses one statically (see Section 3.1).

g `early=` conservative static equality of dynamic values, see Section 3.2.

4.2.5 Static Extensions

Previously [Draves96], the code pointer in a static closure (from a function or a continuation) was represented with a metastatic extension. The binding times in such an extension are fixed, which restricted the use of polyvariance and required additional annotation. Specifically, when you made a static stack frame you had to give the binding time of the return value. This conflicts with the polyvariance in the binding times needed to support bit-addressing.

Furthermore, the implementation of late normalization of cyclic values (Section 3.3) requires being able to identify all cyclic values in the static state of the compiler.

I solved these together by changing the representation of extension to include the original code pointer and the binding time of the first argument (i.e. the frame inside the closure). So when the compiler is running and the static extension is called, the rest of the binding times are available. At this point we call cogen to get the metastatic extension, and then jump to that. Because we memoize on binding times, this results in lazy compiler generation; cogen is called at compile time. In this way, Nitrous is a hybrid between offline and online systems like [Sperber96].

This was a big change conceptually, but a small change in terms of the code and what actually happens.

There are two places where static extensions are created: static recursions and higher-order values.

Because the stack is an explicit argument, when `cogen` encounters a static recursion the same label will eventually appear on two stack frames. In theory, because $(\widetilde{const}\ x) \sqcup (\widetilde{const}\ y) = \mathtt{S}$, when this loop is collapsed the metastatic continuations would be lifted to static, thereby converted to extensions and forming a control stack in the compiler[2]. However, to simplify the implementation `cogen` uses a special lift that makes a static extension:

instruction ::= ... | `(lift` *v* `stack)`

This saves *v* and its binding time in a static extension, and sets its binding time to *stack-bt*.

For example, consider specialization of `append`[3]. The source with annotations appears in Figure 4.20. When `cogen` is called to create `append(D S D)`, it calls itself recursively to create `append(`*`stack-bt`*` S D)`, causing a recursive call to create `append(`*`stack-bt`*` S D)` again. Since this binding-time pattern is now in the memo-table, the recursion terminates. There are two extensions made from `append`. The difference is the jump to the continuation in the base case. In the first the jump target is dynamic, in the second it is static (but not constant). If the list is not zero length then when we call `append(`*`stack-bt`*` S D)` with the static values, then each time the compiler returns it is a call to a static extension. The first return creates `cont((`$\widetilde{close}$` cont (`$\widetilde{list}$` `*`stack-bt`*` S D)) D)`. Subsequent returns except for the last one request the same binding-time pattern, which is in the memo-table. Figure 4.21 illustrates this series of events. Note that there are two versions of each extension. `Cogen` could avoid producing the (probably over-) specialized entry/exit code by checking for the

[2] If instead of forgetting x and y completely, we approximated them with a set, the result would be control-flow analysis based on abstract interpretation [Shivers91].

[3] Only a irregular recursion really requires this, but `append` is easier to understand.

```
append(k l m) {
  if (null? l) (car k)(k m)
  frame = (list k l m)
  cl = (close cont frame)
  lift cl stack
  append(cl (cdr l) m)
}
cont(self r) {
  (k l m) = (cdr self)
  nr = (cons (car l) r)
  lift nr
  (car k)(k nr)
}
```

Figure 4.20: Annotated code for a static recursion.

end of the stack explicitly. Static extensions are also created for source lambda abstractions, see the case `lambda?` in Appendix A.5.

Late normalization of cyclic values requires that all cyclic values be identified and simultaneously normalized at the beginning of the construction of a dynamic code block (after checking the memotable). These values are identified by their binding times. So we need the binding times of the values inside of closures and the stack. This is the other reason.

Shape un/lifting and Sharing

Here we consider lift case 5 from Section 4.2.3 in greater detail. Say `f` has one argument besides itself. Then lifting `(`$\widetilde{close}$ `f` *frame*`)` $\rightarrow$ `D` creates a call to the extension `f((`$\widetilde{close}$ `D` *frame*`) D)`. The extension is used to fold the static part of *frame* into `f`. The problem is, according to its binding-time pattern, the extension expects the dynamic part of the frame to be passed in separate registers (because of variable splitting), but at the call site the value is pure dynamic, so they are all stored in one register.

Nitrous uses special code at the call site to save (lift) the shape, and in the extension wrapper to restore (unlift) the shape. This code optimizes the transfer by only saving each register once, even if it appears several times in the shape (typically a lexical environment appears many times, but we only need to save

```
append(D S D)
    append(stack-bt S D)
      *append(stack-bt S D)

  cont((close cont (list stack-bt S D)) D)
 *cont((close cont (list stack-bt S D)) D)
 *cont((close cont (list stack-bt S D)) D)
  ...
  cont((close cont (list D S D)) D)
```

Figure 4.21: Sequence of events when building a static recursion. A * indicates a hit on the memo-table.

the subject-values once). The same optimization prevents a normal jump from passing the same register more than once.

Dynamic Control Stacks

How do we extract the dynamic stack of a Sal program from the stack in the Sal interpreter? Say `cogen` encounters `do-call(`*stack-bt* `S S` *alist-bt*`)` (see Figure 4.22). When `cl` is lifted we compute `cont((`$\widetilde{close}$ `S (`$\widetilde{list}$ *stack-bt* `S)) D)`. We want to generate a procedure call where `cont` jumps to `apply`, so inlining is disabled and we lift the stack (`k`) to `D`, invoking lift base case 4. The problem is the extension was made assuming the code pointer would be static, but now it will be dynamic. The unlift code inserts an additional cdr to skip the dynamic value, thus allowing an irregular stack pattern to be handled.

Representations of Cyclic Values

If cyclic values as described in Sections 3.1 are applied to the filter example from Chapter 5, a limitation is revealed. The problem is the addresses are cyclic values so before you can load a word the address must be lifted, resulting in a dynamic multiplication and addition. One way to solve this is to use a different representation: rather than use *q* as the dynamic value, one can use *bq*. This is *premultiplication*. On most RISC architectures the remaining addition can be folded into the load instruction.

The disadvantage of premultiplication is that multiplication and division can no longer maintain sharing information. Which representation is best depends

```
do-call(k fn exp env) {
  frame = (list k fn)
  cl = (close cont frame)
  lift cl stack
  eval(cl exp env)  a
}

cont(self arg) {
  (k fn) = (cdr self)
  lift k
  $inline = #f
  apply(k fn arg)
}
```

Figure 4.22: annotated code to produce a dynamic stack frame. Notes: *a* the call to evaluate the argument is inlined.

on how the value is used. A constraint system should suffice to pick the correct representation.

4.3 Sal

The name "Sal" is derived from "sample language". There are two versions of Sal, one in Nitrous, and is the input to the Simple system. This section concerns the former. After giving the syntax of the language, this section describes its implementation with Nitrous. Sal is basically a recursive equations language with threaded store, `let`, `lambda`, syntactic sugar, and annotations (that is, information ignored by the ordinary semantics of the language, but essential to `cogen`). Figure 4.23 gives a grammar for the syntax. The semantics are obvious except as noted:

a first-order call or primitive application, the name refers to a definition.

b higher-order call, the first expression must evaluate to a lambda.

c access the threaded store.

d generates an assignment to the special `$inline` variable.

See Appendix A.5 for the source code for the interpreter. That this is `root` code makes it difficult to read; if written in direct-style with high-level syntax, it would be ordinary. It could easily be bootstrapped and implemented in Sal itself.

The compiler generated by `cogen` compiles Sal programs to `root` programs. It works in one-pass, performing CPS conversion, closure conversion using a linked representation, and executes tail-recursion without the stack. It generates some duplicate code unnecessarily, and is non-optimizing. Conversion to continuation-passing style is a standard result in PE [Danvy96], but the others are new.

It is possible to feed this generated code back into `cogen`. This is how the benchmarks of Section 5.2 were done.

Examples of residual `root` code generated by this compiler appears in Appendex A.6. They show the duplication problems, as well as the residual annotations required for two-level specialization.

```
prog ::= (defn+)
defn ::= (name (v*) exp)
       | (name prim)

prim ::= Scheme procedure
v ::= variable
name ::= variable
bt ::= binding time

exp ::= variable
      | constant
      | (name exp*)                  ; a
      | (lambda (v*) exp)
      | (exp+)                       ; b
      | (if exp exp exp)
      | (let ((v exp)+) exp)
      | (and exp exp)
      | (or exp exp)
      | (begin exp+)
      | (lift exp)
      | (lift exp bt)
      | (set-base exp exp)
      | (case exp (constant exp)+)
      | (destruct (v+) exp exp)
      | (get-memory)                 ; c
      | (set-memory! exp)            ; c
      | (no-inline)                  ; d
```

Figure 4.23: The grammar for the syntax of Sal. See text for notes.

Chapter 5

Benchmarks

This chapter reports on the building and measurement of several media processing kernels. First I ignore automatic specialization and compare the manual implementation of the alternative strategies (interpreted, buffered, and specialized) from Section 1.1. Then I present the benchmark data of the residual code from Nitrous and Simple, the two implementations of bit-addressing. A partial implementation (cyclic values only) is described briefly in Appendix A.7.

Except as noted below, the methodology is as follows: I use GCC v2.7.2 with the `-O1` option. Although I collected some data with the `-O2` option, it was not significantly different, so I discarded it.

Each of the examples was run for 1000 iterations, while real elapsed time was measured with the `gettimeofday` system call. The whole suite was run five times, and the best times were taken.

The legend of each bar-chart indicates which data-sets come from which machines. Four machines were used to collect the data:

R4k SGI Indigo 2 with 150Mhz R4400 running IRIX 5.3.

P5 IBM Thinkpad 560 with 133Mhz Pentium running Linux 2.0.27.

The height of a bar indicates the ratio of the two execution times specified in the caption.

5.1 Manual

The graphs in Figure 5.1 compare specialization to interpretation and buffering. The general, unspecialized versions here are built on `load_sample` directly (like

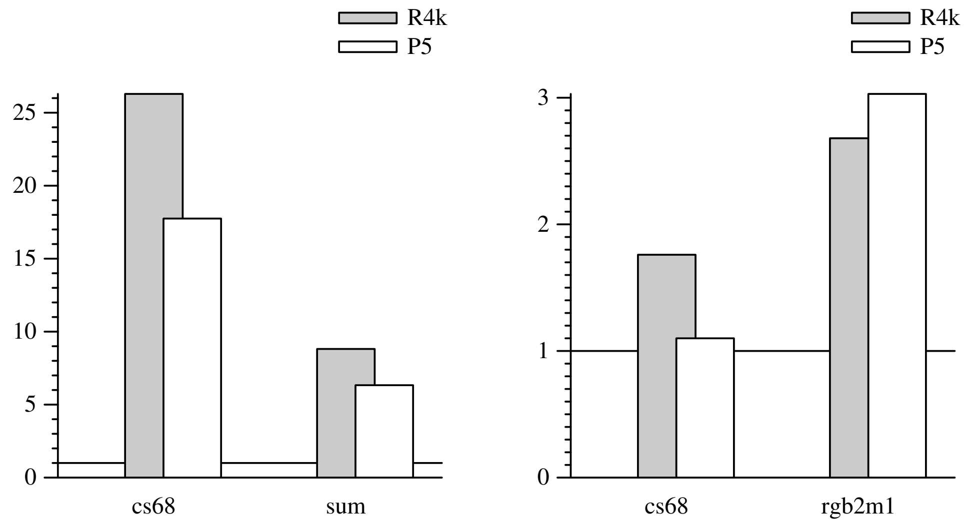

Figure 5.1: The graph on the left shows the execution time of general code normalized to specialized code. The right shows buffered code normalized to specialized code.

`sum` in Figure 3.2) rather than with the interface in Figure 5.4. As expected, the unspecialized programs run many times slower than their specialized counterparts.

The buffered code uses typical operations on vectors of whole-word integers such as multiply every member by a scalar, or add one vector into another. It also uses encode and decode routines that copy a signal from its packed representation into a whole-word vector. These routines have a special case for bytes, otherwise they call `load/store_sample`. The vectors are 200 words long. Cs68 uses three passes, and rgb2m1 uses ten. The code appears in Appendix A.4.

[ThoDa95] analizes buffered audio synthesis when all the data fit into an on-chip cache. They find (and this is corroborated by anecdotal evidence) that buffering reduces bandwidth by about 30%, and the cost is fairly independent of the number of passes over the buffers. This data set indicates that the cost can be more than 200% and may grow with the number of passes. The lower overheads may result from using application-specific vector primitives. In other words, some amount of manual specialization has already taken place.

Next, I compare using `char*` pointers and reading individual bytes to reading whole words and using shift and masks (as bit-addressing does). Figure 5.2 shows that reading words runs slightly faster despite using more instructions. I expect

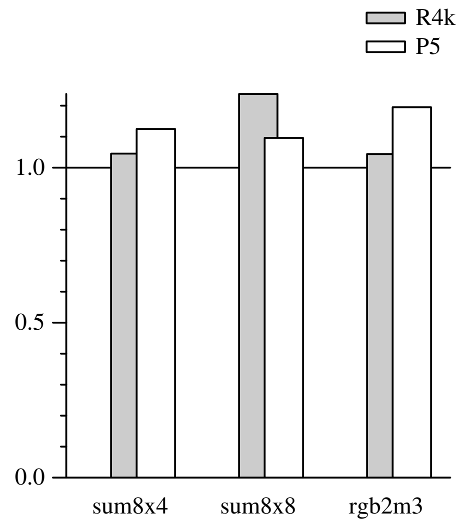

Figure 5.2: Execution time of byte-loads normalized to word-loads with shifts and masks.

that these results are rather dependent on the microarchitecture. The sum8x4 and sum8x8 benchmarks add up the members of a contiguous vector of bytes. The byte-loop is unrolled four and eight times, respectively (the word version is always unrolled four times). The rgb2m3 benchmark is explained below.

5.2 Nitrous

This section presents the a small data-set from the Nitrous system. It compares bit-addressing and automatic specialization to hand-written code. The programs of Figure 5.3 were implemented in `root`, including the cache and signal library. All the residual `root` code is translated to one large C procedure by using GCC's indirect-goto extension for jumps. These examples were run on 2500 bytes of data instead of 4000.

nybble4 sum vector of 4-bit samples

nybble12 sum vector of 12-bit samples

filter3 filter word-vector with kernel of size three.

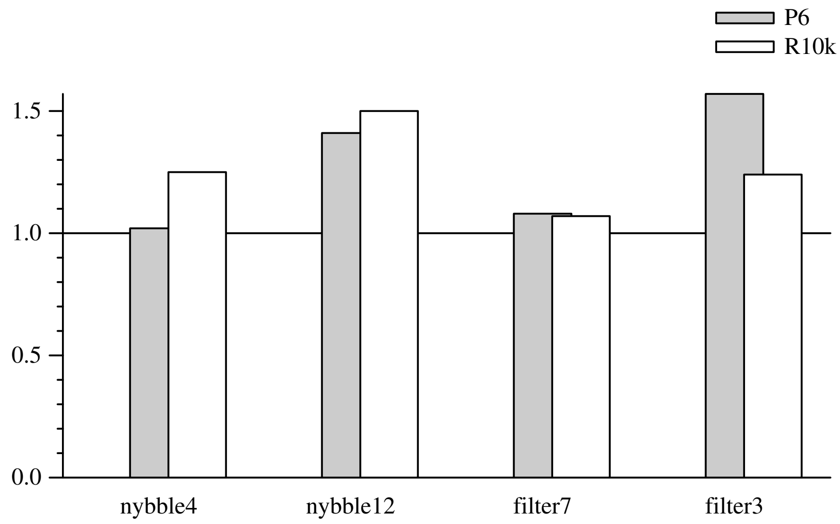

Figure 5.3: Execution time of `root` code automatically specialized by Nitrous normalized to time of hand-specialized code.

filter7 filter word-vector with kernel of size seven.

5.3 Simple

In order to scale-up the examples I built *Simple*, an online specializer that does not use shift/reset or continuations and restricts dynamic control flow to loops (i.e. sum and arrow types are not fully handled). All procedure calls in the source programs are expanded, but the input language is extended with a while-loop construct that may be residualized:

Exp ::= ... | `loop` Var Exp Exp Exp Exp

which is equivalent to the definition and application of the recursive procedure:

```
let fun G Var = if Exp then Exp else G Exp
in G Exp end
```

The loop construct is specialized according to the dynamic conditional heuristic and memoization: it is inlined until the predicate is dynamic, then the loop is entered and unrolled until the predicate is dynamic again. At this point, the static part must match the static part at the previous dynamic conditional.

As described in Chapter 2, because Simple does not perform let-insertion, it may duplicate or forget side-effects. Since GCC performs common subexpression elimination, most but not all of the duplication is eliminated.

The input to Simple is also the Sal language, but with an ML-like syntax and the above restrictions. Examples of its use appear in Figures 1.5 and 1.14, and Appendix A.3. The information is specified with unique names (constants really) created with a quote syntax. The names are ordinary first-class values. The `clone` primitive copies a data-structure including sharing information and creates a new value with the same pattern of sharing internally, but that doesn't share with anything else.

Simple differs from Nitrous in several places. In Nitrous, a generated compiler knows the shapes of the dynamic values, which are the names of their location in the residual code. Early equality works by comparing these locations. In Simple, a dynamic value is associated with an expression. Early equality works by textual equality of these expressions.

Simple does not provide premultiplied cyclic values (see Section 4.2.5). Nor does it use late normalization (see Section 3.3).

Simple is implemented in SML/NJ without concern for execution speed. The specializer requires fractions of a second to produce the examples presented here.

5.3.1 Example

The main example built with Simple is an audio/vector library. It provides the `signal` type, constructors that create signals from scalars or sections of memory, combinators such as creating a signal that is the sum of two other signals, and destructors such as `copy` and `reduce`. The vector operations are suspended in constructed data until a destructor is called. Figure 1.13 contains a graphical representation of this kind of program.

Figure 5.4 gives the signature for part of the library. The semantics and implementation are mostly trivial, most of the code appears in Appendix A.2. One exception is that operations on multiple signals use a conjunction of the end-tests, as described in Section 3.1.1. Consider the end-test of an infinite signal such as a constant or noise (a signal of pseudo-random numbers). It should return true because these signals can end anywhere, rather than returning false because they never end.

```
sig
  type samp
  type signal
  type baddress
  type binop = samp * samp -> samp

  fun vector_signal: baddress * baddress
            * int * int -> signal
  fun constant: samp -> signal

  fun map: (samp -> samp) * signal -> signal
  fun map2: binop * signal * signal -> signal
  fun delay1: signal * samp -> signal
  fun scan: signal * samp * binop -> signal
  fun lut: baddress * signal -> signal
  fun sum_tile: samp * signal * int -> signal

  fun copy: signal * signal -> unit
  fun reduce: signal * binop * samp -> samp

  fun filter: signal * (samp list) * (samp list)
               -> signal
  fun fm_osc: signal * int * baddress * int *
              signal * int -> signal
end
```

Figure 5.4: Signature for signal library.

This delay operator returns a signal of the same length as its input, thus it loses the last sample of the input signal. The other possibility (that it returns a signal one longer) requires sum-types because there would be a dynamic conditional in the `next` method.

The filter combinator is built out of a recursive series of delays, maps, and binops. Similarly, the wavetable combinator is built from simpler components. `Lut` transforms a signal by looking up the samples in an array in memory where the translations are stored one per word. A general version that used another signal as the look-up-table would require a `translate` method, not shown here. `Translate` is also required for the 2D graphics interface, but I have not yet finished with it so I cannot present it here.

A feedback combinator can be made in a higher-order language. Show the code.

With this system, interleaved vectors are stored in the same range of memory, Figure 3.1(c) is an example of three interleaved vectors. With an ordinary vector package, if one passes interleaved vectors to a binary operation, then each input word is read twice. An on-chip hardware cache makes this second read inexpensive, but with the software cache the situation is detected *once* at code-generation time. Specialization with sharing can replace a cache hit with a register reference.

Collapsible protocol languages such as [HaRe96, ProWa96] can handle more advanced control flow (our signals are push or pull, not both), but these systems do not address bits. The same is true of synchronous real-time languages like Signal [GuBoGaMa91] and Lustre [CPHP87]. Their compilers are mostly concerned with deriving global timing from local behavior.

Past work in bit-level processing has not emphasized implementation on word machines. Although C allows one to specify the number of bits used to store a field in a structure, it does not provide for arrays of bits. The hardware design language VHDL [IEEE91]) allows specification of signals of particular numbers of bits, but lacks a compiler that produces fast code.

There are two groups of examples, the audio group (Figure 5.6) and the video group (Figure 5.5). The audio group uses 2000-byte buffers and 16-bit signals; the video group uses 4000-byte buffers and mostly 8-bit signals.

The graphs show the ratio of the execution time of the code generated by Simple to manually written C code. The data indicate that, with some exceptions, the runtimes are comparable. I suspect the exceptions result from aliasing preventing GCC's CSE.

In the audio group, this code was written using `short*` pointers and processing one sample per iteration. In the video group, the code was written using whole-word memory operations and immediate-mode shifts/masks. Some of the code appears in Appendix A.4.

Some of the static information used to create the specialized loops appears in Appendix A.3. These are generally arguments to the interpreter `copy`, which is used for all the audio examples. The video examples also use `copy`, except `iota`, `sum`, and `sum12`.

The audio examples operate on sequential aligned 16-bit data unless noted otherwise:

inc add 10 to each sample.

add two signals to form a third.

filter2 filter with kernel width 2.

filter5 filter with kernel width 5. The manual code doesn't unroll the inner loop over the kernel.

fm1 a one oscillator wave-table synthesizer.

fm2 a one-in-one oscillator wave-table synthesizer.

lut a look-up table of size 256. The input signal is 8-bits per pixel.

sum all the samples in the input

wavrec an wave-table synthesizer with feedback.

The video examples operate on sequential aligned 8-bit data unless noted otherwise:

copy no operation.

gaps destination signal has stride 16 and size 8.

cs68 converts binary to ASCII by reading a six-bit signal and writing eight.

cs86 ASCII to binary by reading eight and writing six.

iota fills bytes with 0, 1, 2, ...

sum as in Figure 3.2, specialized as in Figure 3.3

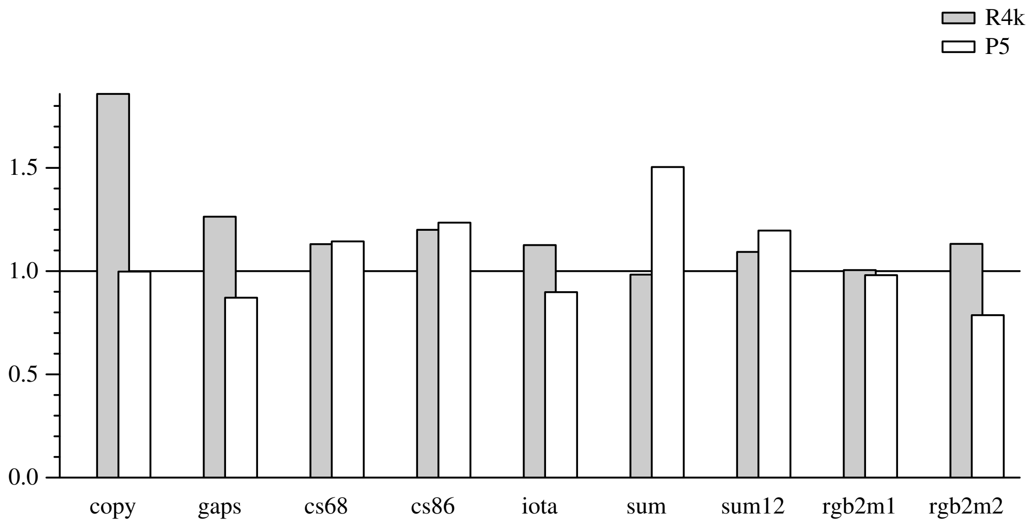

Figure 5.5: Video group. Execution time of automatically specialized by Simple normalized to time of hand-specialized code.

sum12 a twelve-bit signal, as in Figure 3.4.

rgb2m1, rgb2m2, rgb2m3 convert color to monochrome. The input pixels are layed out as in Figures 1.1, 1.2, and 1.3, respectively. The output data are sequential bytes in each case.

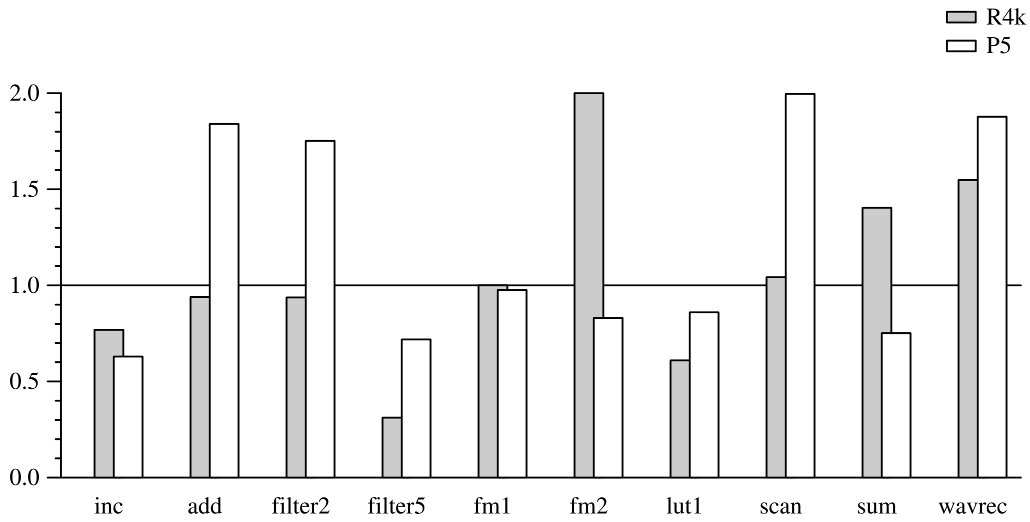

Figure 5.6: Audio group. Execution time of code automatically specialized by Simple normalized to time of hand-specialized code.

Chapter 6

Conclusion

I have shown how to apply specialization to problems in media-processing. The idea has been implemented and the benchmarks show it has the potential to allow programmers to write and type-check very general programs, and then create specialized versions that are comparable to hand-crafted C programs.

The fundamental idea is that semantics-based compiler generation is a portable, easy-to-use interface to run-time code generation. This improves on performing traditional macro-expansion at run-time in three ways: first a program can be written and debugged with specialization disabled, that is, without RTCG. Second, variable capture and quoting errors are impossible to make. Third, the code generators can be specialized themselves, resulting in low-overhead RTCG.

Furthermore, I proposed introducing knowledge of the linear-algebraic properties of integers into the specializer instead of treating them as atoms. The programmer can write high-level specifications of loops, and generate efficient implementations with the confidence that the system will preserve the semantics of their code. By making aliasing and alignment static, many of the operations normally performed by a hardware cache at runtime can be done at code generation time.

Of course, problems remain. The next section describes the difficulties I encountered as a programmer while building the benchmark examples. I then take a step back, and assess the current usability of the systems, and the prospects for improvement.

6.1 Pitfalls and Prospects

Sometimes specializing programs with cyclic values produces excessive special cases. Say I want to make a routine `F` that manipulates an arbitrary byte-vector in memory. As a bit-address, a byte-pointer is a cyclic value zero mod eight. So I write

```
fun sumb (p, q) =
  let val s = vector_signal(set_base(p,32),
                    set_base(q,32), 8, 8)
  in reduce(s, plus, 0)
  end
```

$$\mathtt{F} = \mathcal{S}\ \mathtt{sumb}\ [\mathtt{p} \mapsto \langle 8\ \boxed{\mathrm{pb}}\ 0\rangle\quad \mathtt{q} \mapsto \langle 8\ \boxed{\mathrm{qb}}\ 0\rangle]$$

and indeed `F` does exactly what I want, and is as fast as I expect. But it unnecessarily contains four copies of the inner loop, one for each possible alignment of the terminating pointer (`q`). A `set_base` of just the initial pointer results in one copy of the inner loop. I should be able to get the smaller code by writing

```
fun sumb (p, q) =
  let val evend = q - (q%32)
      val s0 = vector_signal(set_base(p,32), evend, 8, 8)
      val s1 = vector_signal(evend, set_base(q,32), 8, 8)
  in reduce(s0, op_plus, reduce(s1, op_plus, 0))
  end
```

Nitrous frequently fails to terminate because it tries to generate either infinite or exponential quantities of code. This is not surprising considering that it was designed to err on the side of propagating too much information. For example, because of the code duplication in dynamic loops (see Section 4.2.5), nesting loops results in code whose size is exponential in the levels of nesting.

A more interesting source of code explosion is the cache. The inclusion of the partially-dirty cache lines in particular may produce a large number of static states. Irregular patterns of writes can create this (consider Figure 6.1). Even if the size and stride of the signal are simple (say both are eight), then the masks in the cache still suffer from exponential explosion because each byte could be either clean or dirty. One way to reduce the expansion is to flush the cache to return it to a known state. This is like defining a procedure-calling convention for the cache.

Section 3.1.2 presents an example of infinite specialization resulting from too much inlining. The need for the `dcall` annotation was not anticipated.

```
fun make_write_head p stride size =
  (fn s_put => (fn v => store_sample p size v)
    | s_shift => (make_write_head (p+d*stride) stride size))
```

Figure 6.1: Irregular writes.

```
fun make_counter from by =
 fn s_end => true
  | s_next => make_counter (from+by) by
  | s_get => from

val count_by by = make_counter (lift 0) by

val two_counters = map2 op_plus (count_by 2) (count_by 3)
```

Figure 6.2: With Simple, any loop over `evens` will fail.

An unexpected problem results from the combination of Simple's restriction of dynamic control flow to loops and sharing. The code in Figure 6.2 (in the style of Appendix A.2) exhibits the problem. The two dynamic values in `two_counters` (two instances of `(lift 0)`) are equal and shared. If `two_counters` is passed to `reduce` then after the first iteration the dynamic values diverge, so the sharing is lost. But in Simple, the static state at the top and bottom of a loop must match.

In summary, none of the implementations of bit-addressing is currently practical. Nitrous is too slow, and Simple is too restricted. The Similix implementation is promising because, like the Simple system, its specializers run in fractions of a second, but without the restrictions of Simple. None of these systems has a fast enough backend to generate code at runtime. But I believe that application of standard engineering practice to the ideas of this thesis would result in systems useful to expert programmers. I now outline such systems, and the difficulties they will face.

Modifying an existing second-generation polyvariant specializer like PGG (or its ML equivalent, if available) is a natural choice. Cyclic integers are easy to add, though more cases than are shown here may be useful. Sharing and conservative equality appear to pose no difficulty, though the global state traversal in late normalization might be a problem. The function of late normalization is to make the early equality smarter at very little expense at code generation time, and so may be pointless if code generation is slow. The output would be compiled with an ordinary Scheme (or ML) compiler. The advantage of this approach is leveraging existing backends and frontends. The disadvantage would be relatively slow code generation (high $\overline{\sigma}$), and slow residual code (not as fast as GCC).

Alternatively, a new specializer (frontend) may be developed. The system could work with an existing backend with a strongly typed intermediate langauge and fast code generation (e.g. the JVM), or a new backend could be derived for an existing system (e.g. TIL2 modified to reduce $\overline{\sigma}$).

Both of these approaches would probably solve the basic problems of Nitrous (slow execution and excessive code duplication) and Simple (restricted languages). But any system based on partial evaluation has a basic problem: debugging the binding times. A mechanism that might help is assertions (such as "this variable should be static"), where the error message in case of violation includes the cascade of guilty variables. Manual placement of some lifts and other tweeks to adjust binding times and inlining seem inevitable. However, I suspect that using profile data from execution to guide specialization could significantly outperform the dynamic conditional heuristic.

This leaves us with the intrinsic problems of implementing bit-addressing as described here:

- When should the cache be initialized (flushed)? This is a special case of when to lift. How many entries should allocated? The current strategy of flushing after a loop sometimes preserves too much information (Figure 6.1 above). Perhaps some signals should be uncached. Perhaps a heuristic can be developed.

- How can we handle sharing information in a modular way? How can we access a specializer from a strongly typed language (recall the different arities of `rgbm1` and `rgbm2` in Figure 1.5). Perhaps the top-level procedure returned by specializer should not have redundant arguments removed, thus permitting a typable external interface.

- How much can we do without negative may-alias info? I expect this invariant becomes difficult to maintain in a large system.

- How can array-bounds checks be eliminated? All the examples in this chapter used unsafe arrays, though this does not scale to large systems. Good compilers can eliminate many of these checks, and I think that RTCG exposes many more. As noted in Section 3.6, the natural way to address this is with generalized partial computation. But GPC is dependent on a theorem prover. We assume that some form of annotations (or interactivity) to guide the prover will be necessary. The Extended Static Checking (ESC) framework [Detlefs96] may prove suitable.

Until more experience with better implementations of bit-addressing is collected, these plans and analyses will remain rather speculative.

Bibliography

[Andersen94] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. DIKU 1994.

[ANSI90] ANSI. *ANSI/ISO 9899-1990: Programming Languages - C*. American National Standards Institute 1990.

[ApMa91] Andrew W Appel, David B MacQueen. Standard ML of New Jersey. *3rd Symposium on Programming Language Implementation and Logic Programming*, 1991.

[Appel92] Andrew Appel. *Compiling with Continuations*. Cambridge University Press 1992.

[ASeUl86] A V Aho, R Sethi, J D Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley 1986.

[BiWe93] Lars Birkedal, Morten Welinder. Partial Ealuation of Standard ML. DIKU-TR-93-22.

[BoDa91] A Bondorf, O Danvy. Automatic Autoprojection of Recursive Equations with Global Variables and Abstract Data Types. *Science of Computer Programming*, 16:151-195.

[BoDu93] Anders Bondorf, Dirk Dussart. Handwriting Cogen for a CPS-Based Partial Evaluator. *Partial Evaluation and Semantics-Based Program Manipulation*, 1994.

[Bondorf92] Anders Bondorf. Improving binding times without explicit CPS-conversion. *ACM Conference on Lisp and Functional Programming*, 1992.

[Carl96] Stephen Carl. *Syntactic Exposures - A Lexically-Scoped Macro Facility for Extensible Compilers*. University of Texas at Austin 1996.

[ChaBleFi91] Siddhartha Chatterjee, Guy E Blelloch, Allan L Fisher. Size and Access Inference for Data-Parallel Programs. *Conference on Programming Language Design and Implementation*, 1991.

[CHNNV96] Charles Consel, Luke Hornof, Francois Noël, Jacque Noyé, Nicolae Volanschi. A Uniform Approach for Compile-Time and Run-Time Specialization. *Dagstuhl Workshop on Partial Evaluation*, 1996.

[CoDa89] Charles Consel, Olivier Danvy. Partial Evaluation of Pattern Matching in Strings. *Information Processing Letters*, 30:79-86.

[CoDa98] Charles Consel, Olivier Danvy. *Partial Evaluation in Procedural Languages*. MIT Press 1998.

[Consel88] Charles Consel. New Insights into Partial Evaluation: The Schism Experiment. *European Symposium on Programming*, 1988.

[Consel90] Charles Consel. Binding Time Analysis for Higher Order Untyped Functional Languages. *ACM Conference on Lisp and Functional Programming*, 1990.

[Consel93] Charles Consel. Polyvariant Binding-Time Analysis For Applicative Languages. *Partial Evaluation and Semantics-Based Program Manipulation*, 1993.

[CPHP87] P Caspi, D Pilaud, N Halbwachs, J A Plaice. Lustre: A Declarative Language for Programming Synchronous Systems. *Principles of Programming Languages*, 1987.

[Danvy96] Olivier Danvy. Type-Directed Partial Evaluation. *Principles of Programming Languages*, 1996.

[DaPfe96] Rowan Davies, Frank Pfenning. A Modal Analysis of Staged Computation. *Principles of Programming Languages*, 1996.

[Detlefs96] David Detlefs. An Overview of the Extended Static Checking System. *Workshop on Formal Methods in Software Practice*, 1996.

[Deutsch94] Alain Deutsch. Interprocedural May-Alias analysis for pointers: Beyond *k*-limiting. *Conference on Programming Language Design and Implementation*, 1994.

[Draves95] Scott Draves. Lightweight Languages for Interactive Graphics. CMU-CS-95-148.

[Draves96] Scott Draves. Compiler Generation for Interactive Graphics using Intermediate Code. *Dagstuhl Workshop on Partial Evaluation*, 1996.

[Draves97] Scott Draves. Implementing Bit-addressing with Specialization. *International Conference on Functional Programming*, 1997.

[EngKaOT95] Dawson Engler, M Frans Kaashoek, James O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. *Symposium on Operating Systems Principles*, 1995.

[EnHsKa95] Dawson Engler, Wilson Hsieh, M Frans Kaashoek. 'C: A Language for High-Level, Efficient, and Machine-independent Dynamic Code Generation. *Conference on Programming Language Design and Implementation*, 1995.

[FriWa84] Daniel P Friedman, Mitchell Wand. Reification: Reflection without Metaphysics. *ACM Conference on Lisp and Functional Programming*, 1984.

[FuNoTa91] Yoshihiko Futamura, Kenroku Nogi, Aki Takano. The essence of generalized partial computation. *Theoretical Computer Science*, 90(1):61-79.

[Futamura71] Y Futamura. Partial evalutaion of computation process - an approach to a compiler-compiler. *Systems, Computers, Controls*, 2:45-50.

[GluJo94] Robert Glück, Jesper Jørgensen. Generating Optimizing Specializers. *IEEE Computer Society International Conference on Computer Languages*, 1994.

[GluJo95] Robert Glück, Jesper Jørgensen. Efficient Multi-Level Generating Extensions for Program Specialization. *Programming Language Implementation and Logic Programming*, 1995.

[GoJo91] Carsten K Gomard, Neil D Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1:21-69.

[GoJoSte96] James Gosling, Bill Joy, Guy Steele. *The Java Language Specification*. Addison-Wesley 1996.

[Graham94] Paul Graham. *On Lisp: Advanced Techniques for Common LISP*. Prentice-Hall 1994.

[GuBoGaMa91] P le Guernic, M le Borgne, T Gauthier, C le Maire. Programing Real-Time applications with Signal. *Proceedings of the IEEE*, 79(9):1305-1320.

[HaRe96] Mark Hayden, Robbert van Renesse. Optimizing Layered Communication Protocols. Cornell-TR96-1613.

[Haynes93] Christopher Haynes. Infer: A Statically-typed Dialect of Scheme. Indiana-CS-TR-93-367.

[Henglein91] Fritz Henglein. Efficient Type Inference for Higher-Order Binding-Time Analysis. *International Conference on Functional Programming Languages and Computer Architecture*, 1991.

[HePa90] John L Hennessy, David A Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann 1990.

[HiDyBru92] Robert Hieb, Kent Dybvig, Carl Bruggeman. Syntactic abstraction in scheme. Indiana University TR #355, 1992.

[IEEE91] IEEE. *IEEE Standard 1076: VHDL Language Reference Manual*. IEEE 1991.

[James1882] William James. Subjective Effects of Nitrous Oxide. *Mind*, Volume 7.

[JHHPW93] Simon L Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, Philip Wadler. The Glasgow Haskell compiler: a technical overview. *UK Joint Framework for Information Technology (JFIT)*, 1993.

[JoGoSe93] Neil D Jones, Carsten K Gomard, Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall 1993.

[Johnson75] Stephen C Johnson. YACC - Yet Another Compiler-Compiler. Bell Labs 1975.

[Jones88] Neil D Jones. Automatic Program Specialization: a Re-examination from Basic Principles. *Partial Evaluation and Mixed Computation*, 1988.

[Jones91] Neil D Jones. Efficient Algebraic Operations on Programs. *Algebraic Methodology and Software Technology*, 1991.

[JoSche86] Ulrik Jørring, William Scherlis. Compilers and Staging Transformations. *Principles of Programming Languages*, 1986.

[JoSeSo85] Neil D Jones, P Sestoft, H Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. *Rewriting Techniques and Applications, Dijon, France*, 1985.

[KeEgHe91] D Keppel, S J Eggers, R R Henry. A Case for Runtime Code Generation. UW-CSE-91-11-04.

[KFFD86] Eugene Kohlbecker, Daniel Friedman, Matthias Felleisen, Bruce Duba. Hygienic Macro Expansion. *ACM Conference on Lisp and Functional Programming*, 1986.

[Lee89] Peter Lee. *Realistic Compiler Generation*. MIT Press 1989.

[LeLe96] Peter Lee, Mark Leone. Optimizing ML with Run-Time Code Generation. *Conference on Programming Language Design and Implementation*, 1996.

[Leroy92] Xavier Leroy. Unboxed objects and polymorphic typing. *Principles of Programming Languages*, 1992.

[MiHraCru94] Melanie Mitchell, James Crutchfield, Peter Hraber. Evolving Cellular Automata to Perform Computations: Mechanisms and Impediments. *Physica D*, 75:361-391.

[MiToHa90] Robin Milner, Mads Tofte, Robert Harper. *The Definition of Standard ML*. MIT 1990.

[Mogensen89] Torben Mogensen. *Binding Time Aspects of Partial Evaluation*. DIKU 1989.

[Mosses78] Peter Mosses. SIS, a Compiler-Generator System using Denotational Semantics. Aarhus-TR-1978.

[Mossin93] Christian Mossin. Partial evaluation of General Parsers. *Partial Evaluation and Semantics-Based Program Manipulation*, 1993.

[MuVoMa97] Gilles Muller, Eugen-Nicolae Volanschi, Renaud Marlet. Scaling up Partial Evaluation for Optimizing the Sun Commercial RPC Protocol. *Partial Evaluation and Semantics-Based Program Manipulation*, 1997.

[PoEnKa97] Massimiliano Polleto, Dawson Engler, M Frans Kasshoek. tcc: A System for Fast, Flexible, and High-level Dynamic Code Generation. *Conference on Programming Language Design and Implementation*, 1997.

[ProWa96] Todd Proebsting, Scott Watterson. Filter Fusion. *Principles of Programming Languages*, 1996.

[PuMaIo88] Calton Pu, Henry Massalin, John Ioannidis. The Synthesis Kernel. *Computing Systems*, 1988.

[R4RS] William Clinger, Jonathan Rees. Revised^4 Report on the Algorithmic Language Scheme. *LISP Pointers*, IV:1-55.

[Reynolds97] John Reynolds. *Programming Languages Core Course Notes*. unpublished 1997.

[Shivers91] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. Carnegie Mellon University, School of Computer Science 1991.

[SiHoMcA96] Satnam Singh, Jonathan Hogg, Derek McAuley. Expressing Dynamic Reconfiguaration by Partial Evaluation. *Symposium on Field-Programmable Custom Computing Machines*, 1996.

[SoGluJo96] Morten Sørensen, Robert Glück, Neil Jones. A Positive Supercompiler. *Journal of Functional Programming*, 6(6):811-838.

[Sperber96] Michael Sperber. Self-Applicable Online Partial Evaluation. *Dagstuhl Workshop on Partial Evaluation*, 1996.

[SpeThi95] Michael Sperber, Peter Thiemann. The Essence of LR Parsing. *Partial Evaluation and Semantics-Based Program Manipulation*, 1995.

[SSOG93] J Subhlok, J Stichnoth, D O'Hallaron, T Gross. Exploiting Task and Data Parallelism on a Multicomputer. *Principles and Practice of Parallel Programming*, 1993.

[Steele90] Guy Steele. *Common Lisp the Language*. Digital Press 1990.

[SteLe95] A Stepanov, M Lee. The Standard Template Library. Hewlett Packard Labs HPL-95-11.

[Thiemann96] Peter Thiemann. Cogen in six lines. *International Conference on Functional Programming*, 1996.

[ThoDa95] Nicholas Thompson, Roger Dannenberg. Optimizing Software Synthesis Performance. *International Computer Music Conference*, 1995.

[TMCSHL96] D Tarditi, G Morrisett, P Cheng, C Stone, R Harper, P Lee. TIL: A Type-Directed Optimizing Compiler for ML. *Conference on Programming Language Design and Implementation*, 1996.

[Wadler88] Philip Wadler. Deforestation: Transforming Programs to Eliminate Trees. *European Symposium on Programming*, 1988.

[WCRS91] Daniel Weise, Roland Conybeare, Erik Ruf, Scott Seligman. Automatic online program specialization. *International Conference on Functional Programming Languages and Computer Architecture*, 1991.

Appendix A

Code Listings

A.1 Cache Source

A fragment of the cache implementation for the simple system:

```
val W = 32

fun mask b = (1 << b) - 1

fun load_sample (p, b) =
    let wa = p / W
    let ba = p % W
    let w0 = (load_word_cached wa)
    let s0 = (mask b) & (w0 >> ba)
    if ((ba + b) > W)
       (let ub = W - ba
        let w1 = load_word_cached ((p+ub) / W)
        let s1 = (w1 & (mask (b - ub)))
        s0 | (s1 << ub))
     s0

fun flush_line line =
    let (addr, clean, mask, v) = line
    if clean line
    let v2 = (if (0 = mask) v
         (v | (mask & (load_word addr))))
    let line2 = (addr, true, 0, v2)
    store_word(addr, v2)
```

```
fun load_word_cached(addr) =
   let (effects,cache) = get_store
   if (is_pair(cache))
      (lw_loop(cache, (), addr))
      (load_word(addr))

fun lw_loop(cache, prev_cache, addr) =
 if (is_pair cache)
   (let (line, rest) = cache
    let (addr2, clean, mask, v) = line
     if (aliases(addr2,addr))
     (if (clean or (mask = 0))
         (cache_done(prev_cache, rest, addr,
                     true, 0, v))
         (error cannot_cross_streams2))
     (lw_loop (rest, (line, prev_cache), addr)))
    ((flush_line(left prev_cache));
     (let w = (load_word(addr))
     (cache_done ((right prev_cache), (), addr,
                  true, 0, w))))
```

A.2 Signal Interface Source

A fragment of the implementation of the signal interface:

```
fun memory_empty  (start, stop, size, stride) =
    (start = stop)
fun memory_next (start, stop, size, stride) =
    (v_memory, ((start+stride), stop, size, stride))
fun memory_get (start, stop, size, stride) =
    load_sample(start, size)
fun memory_put ((start, stop, size, stride), v) =
    store_sample(start, size, v)

fun constant_empty  k = true
fun constant_next k = (v_constant, k)
fun constant_get k = k
fun constant_put (k, v) = (error)
```

```
fun noise_empty  (state, ia, ic, im) = true
fun noise_next  (state, ia, ic, im) =
    (v_noise, (((lift (ia*state + ic)) % im), ia, ic, im))
fun noise_get   (state, ia, ic, im) = state
fun noise_put (state, ia, ic, im) = (error)

fun bin_empty (op, v, w) =
    ((vec_empty v) and (vec_empty w))
fun bin_next  (op, v, w) =
    (v_bin, (op, (vec_next v), (vec_next w)))
fun bin_get   (op, v, w) =
    (do_op (op, (vec_get v), (vec_get w)))
fun bin_put   ((op, v, w), q) = (error)

fun delay1_empty (h, v) = (vec_empty v)
fun delay1_next  (h, v) =
    (v_delay1, ((vec_get v), (vec_next v)))
fun delay1_get   (h, v) = h
fun delay1_put   ((h, v), q) = (error)

fun scan_empty (op, h, v) = (vec_empty v)
fun scan_next  (op, h, v) = (v_scan, (op, (do_op (h, (vec_get v))),
                                      (vec_next v)))
fun scan_get   (op, h, v) = h
fun scan_put   ((op, h, v), q) = (error)

fun reduce (op, init, vec) =
   loop (v, vec) ((lift init), vec)
        (vec_empty vec)
        ((do_op(op, v, (vec_get vec))),
         (vec_next vec))
        v

fun copy (a, b) =
    loop (a, b) (a, b)
         ((vec_empty a) and (vec_empty b))
         ((vec_put (b, (vec_get a)));
          ((vec_next a), (vec_next b)))
         ()
```

```
fun filter (i, k, pre) =
   if (is_pair k)
      (v_binop, (op_plus,
       (v_map, (op_times, (left k), i)),
       (filter ((v_delay1, ((debug (left pre)), i)),
                (right k), (right pre)))))
       i

val sig16 = (v_memory, ((32*('start)+0), (32*('stop)+0), 16, 16))
val sig16_1 = clone sig16
val sig16_2 = clone sig16

val aligned_bytes = ((32*('start)+0), (32*('stop)+0), 8, 8)
val aligned_6s = ((32*'start1+0), (32*'stop1+0), 6, 6)
```

A.3 Signal Examples

Programs implemented with the signal library.

```
val add = (op_plus, sig16, sig16_1, sig16_2)

val inc = (op_plus, sig16, (v_constant, 10),
           sig16_1)

val filter2 = ((v_bin,
                 (op_plus,
                  (v_delay1,
                   (('first), sig16)),
                  sig16)),
               sig16_2)

val kernel = (1, 2, 4, 2, 1, ())
val prefix = (('a), ('b), ('c), ('d), ('e), ())
val filter5 = ((filter (sig16, kernel, prefix)),
               sig16_1)

val lut1 = ((v_lut, (('buf), sig8)), sig16)

val wavtab1 =  ((v_lut_feedback,
                  (('buf), 1024, 1, 32,
                   ('prev), sig16)),
                sig16_1)
```

```
val fm1 = ((fm_osc ((v_constant, 0), 0,
                    ('buf), 1024,
                    (v_constant, 256),
                    ('init_phase))),
           sig16)

val fm2 = ((fm_osc ((osc (('buf), 1024,
                           (v_constant, 256),
                           ('phase0))), 1,
                    ('buf), 1024,
                    (v_constant, 256), ('phase1))),
           sig16)

val rgb2m_1 = rgb2m (rgba_r, rgba_g, rgba_b, mono8)
val rgb2m_2 = rgb2m (rgb_r, rgb_g, rgb_b, mono8)

val base64_encode = (aligned_6s, aligned_bytes)
val base64_decode = (aligned_bytes, aligned_6s)
```

A.4 Manual Code

A.4.1 Specialized

Baseline hand-specialized C code.

```
int
sum16(short *start, short *stop,
      int sum) {
  while (start != stop) {
    sum += *start++;
  }
  return sum;
}
```

```
void
filter2(short *start, short *stop,
        short *start1, short *stop1) {
  while (start != stop) {
    *start1 = start[0] + start[1];
    start++;
    start1++;
  }
}

void
filter5(short *start, short *stop,
        short *start1, short *stop1) {
  int i, t;
  while (start != stop) {
    t = 0;
    for (i = 0; i < 5; i++)
      t  += start[i];
    *start1 = t;
    start++;
    start1++;
  }
}

int
sum8(int *start, int *stop,
     int sum) {
   int v;
   while(start != stop) {
     v = *start;
     sum += (((v>>0)&255) +
             ((v>>8)&255) +
             ((v>>16)&255) +
             ((v>>24)&255));
     start += 1;
   }
   return sum;
}
```

```
void
iota(int *start, int *stop) {
  int i = 0;
  while(start != stop) {
    *start++ = i | ((i+1)<<8) |
      ((i+2)<<16) | ((i+3)<<24);
    i+=4;
  }
}

void
copy(int *start0, int* stop0,
     int *start1, int* stop1) {
  while (start0 != stop0)
    *start0++ = *start1++;
}

void
gaps(int *start0, int* stop0,
     int *start1, int* stop1) {
  while (start0 != stop0) {
    int v = *start0;
    int b0 = (v>>0)&255;
    int b1 = (v>>8)&255;
    int b2 = (v>>16)&255;
    int b3 = (v>>24)&255;
    int mask = 0xff00ff00;
    start1[0] = (start1[0] & mask)
      | b0 | (b1 << 16);
    start1[1] = (start1[1] & mask)
      | b2 | (b3 << 16);
    start0++;
    start1+=2;
  }
}
```

```
int
sum12(int *start, int *stop) {
  int sum = 0;
  while (start != stop) {
  int w0 = start[0];
  int w1 = start[1];
  int w2 = start[2];
  sum += ((w0 & 0xfff) +
          ((w0 >> 12) & 0xfff) +
          (((w0 >> 24) & 0xff)
           | ((w1 & 0xf) << 8)) +
          ((w1 >> 4) & 0xfff) +
          ((w1 >> 16) & 0xfff) +
          (((w1 >> 28) & 0xf)
           | ((w2 & 0xff) << 4)) +
          ((w2 >> 8) & 0xfff) +
          ((w2 >> 20) & 0xfff));
  start += 3;
  }
  return sum;
}

void
fm1(int *lut, int phase,
    short *start, short *stop) {
  while (start != stop) {
    *start++ = lut[phase>>8];
    phase += 256;
    phase = phase & ((1024*256)-1);
  }
}
```

A.4.2 Buffered

Buffered versions.

```
typedef unsigned int bitp;
```

```
typedef struct {
  bitp start;
  bitp stop;
  int stride;
  int size;
} signal_t;

void decode_signal(signal_t *s, int *v) {
  bitp s0=s->start, s1=s->stop;
  int stride = s->stride;
  int size = s->size;
  if ((stride&7) == 0 &&
      size == 8 &&
      (s0&7) == 0) {
     char *p = ((char *)fixaddr(s0>>5)) + ((s0>>3)&3);
     char *q = ((char *)fixaddr(s1>>5)) + ((s0>>3)&3);
     stride = stride>>3;
     while (p != q) {
        *v++ = *p;
        p += stride;
     }
  } else
     while (s0 != s1) {
        *v++ = load_sample(s0, size);
        s0 += stride;
     }
}
```

```
void encode_signal(signal_t *s, int *v) {
  bitp s0=s->start, s1=s->stop;
  int stride = s->stride;
  int size = s->size;
    if ((stride&7) == 0 &&
      size == 8 &&
      (s0&7) == 0) {
     char *p = ((char *)fixaddr(s0>>5)) + ((s0>>3)&3);
     char *q = ((char *)fixaddr(s1>>5)) + ((s0>>3)&3);
     stride = stride>>3;
     while (p != q) {
        *p = *v++;
        p += stride;
     }
  } else while (s0 != s1) {
    store_sample(s0, size, *v++);
    s0 += stride;
  }
}

void scale_vector(int *v, int s, int n) {
  int i;
  for (i = 0; i < n; i++)
    v[i] *= s;
}

void translate_vector(int *v, int s, int n) {
  int i;
  for (i = 0; i < n; i++)
    v[i] += s;
}

void add_vectors(int *v, int *w, int n) {
  int i;
  for (i = 0; i < n; i++)
    w[i] += v[i];
}
```

```
void divide_vector(int *v, int s, int n) {
  int i;
  for (i = 0; i < n; i++)
    v[i] /= s;
}

rgb2m_buff(signal_t *r, signal_t *g, signal_t *b,
           signal_t *m, int cr, int cg, int cb, int cs) {
  int t0[4000];
  int t1[4000];
  int n = (r->stop - r->stop)/r->stride;
  decode_signal(r, t0);
  scale_vector(t0, cr, n);
  decode_signal(g, t1);
  scale_vector(t1, cg, n);
  add_vectors(t0, t1, n);
  decode_signal(b, t0);
  scale_vector(t0, cb, n);
  add_vectors(t0, t1, n);
  divide_vector(t1, cs, n);
  encode_signal(m, t1);
}

cs68_buff(signal_t *i, signal_t *o) {
  int t0[4000];
  int n = (i->stop - i->stop)/i->stride;
  decode_signal(i, t0);
  translate_vector(t0, 32, n);
  encode_signal(o, t0);
}
```

A.4.3 Interpreted

Interpreted versions.

```
typedef unsigned int uint;
```

```
uint
load_sample(uint addr, int len) {
   int mask = (1 << len) - 1;
   uint *p = fixaddr(addr >> 5);
   int o = addr & 31;
   int r = (*p >> o) & mask;
   if (len + o >= 32) {
      int e = (len + o - 32);
      return r | ((p[1] & ((1 << e) -1 )) << e);
   }
   return r;
}

void
store_sample(uint addr, int len, int v) {
   int mask = (1 << len) - 1;
   uint *p = fixaddr(addr >> 5);
   int o = addr & 31;
   int prev = (*p & ~(mask<<o));
   *p = prev | (v<<o);
   if (len + o >= 32) {
     int e = (len + o - 32);
     store_sample(addr+(32-o), e, (v>>(32-o)));
   }
}

int
sum_reduce(uint from, uint to, int bits, int stride) {
   int sum = 0;
   while (from != to) {
      sum += load_sample(from, bits);
      from += stride;
   }
   return sum;
}
```

```
int
cs68(uint from, uint to, uint start0) {
  while (from != to) {
    store_sample(start0, 8, 32+(load_sample(from, 6)));
    start0 += 8;
    from += 6;
  }
}
```

A.5 SAL interpreter

Most of the implementation of SAL.

```
(code lookup
      (k M var env)
      ((prim more? ,pair? env)
       (if more? ((prim hd ,car env)
                  (prim tl ,cdr env)
                  ,@(make 'lookup-found '(k M var tl) 'cl1)
                  ,@(make 'lookup-lost '(k M var tl) 'cl2)
                  ; this unquote is linking/module level
                  (const assq ,assq-root)
                  (jump assq cl1 cl2 var hd)))
       ; raise exception on meta-level, since we don't have them
       ; here.  like a trap.
       (const msg "variable ~S not bound")
       (prim xx ,error msg var)
       ,@(call 'k '(M var))))

(code lookup-found
      (self val)
      (,@(unmake '(k M var tl) 'self)
       ,@(call 'k '(M val))))

(code lookup-lost
      (self val)
      (,@(unmake '(k M var tl) 'self)
       ; search the next frame
       (const lookup lookup)
       (jump lookup k M var tl)))
```

```
(define break-let-clauses
  (code-rec1
   `((code break-let-clauses
           (k M l rv rc)
           ((prim n ,null? l)
            (if n (,@(call 'k '(M rv rc))))
            (prim hd ,car l)
            (prim tl ,cdr l)
            (prim v0 ,car hd)
            (prim t0 ,cdr hd)
            (prim c0 ,car t0)
            (prim rv^ ,cons v0 rv)
            (prim rc^ ,cons c0 rc)
            (const break-let-clauses break-let-clauses)
            (jump break-let-clauses k M tl rv^ rc^))))))

(code apply
      (k M prog name args)
      ((const apply-loop apply-loop)
       (jump apply-loop k M prog prog name args)))
```

```
(code apply-loop
      (k M prog prog2 name args)
      ((prim more? ,pair? prog)
       (prim no-more? ,not more?)
       (if no-more? ((const msg "procedure ~S not bound")
                     (prim xx ,error msg name)
                     ,@(call 'k '(M name))))
       (prim d ,car prog)
       (prim nm ,car d)
       (prim found-defn? ,eq? nm name)
       (if found-defn? ((prim d1 ,cdr d)
                        (prim d2 ,cdr d1)
                        (prim extern? ,null? d2)
                        (if extern?
                            ((prim code ,car d1)
                             (const apply-extern apply-extern)
                             (jump apply-extern k M code args)))
                        (prim formals ,car d1)
                        (prim exp ,car d2)
                        (const nil ())
                        (lift nil
                              (run ,(lambda (self)
                                      (make-multi-env-bt
                                       (make-env-bt
                                        (find-code 'zip root-zip))
                                       (find-code 'subst-cont
                                                  self)))))
                        (const subst subst)
                        (jump subst k M formals args exp nil prog2)))
       (prim rest ,cdr prog)
       (const apply-loop apply-loop)
       (jump apply-loop k M rest prog2 name args)))

(code expr
      (k M exp env prog)
      ((const expr expr)
       (prim is-var? ,symbol? exp)
       (if is-var? ((const lookup ,lookup-recur)
                    (jump lookup k M exp env)))
```

```
(prim is-const? ,atom? exp)
(if is-const? ((lift exp)
               ,@(call 'k '(M exp))))

(prim hd ,car exp)
(prim tl ,cdr exp)
(const get-memory get-memory)
(prim is-get-mem? ,eq? hd get-memory)
(if is-get-mem? (,@(call 'k '(M M))))

(const if-name if)
(prim is-if ,eq? if-name hd)
(if is-if ((prim e0 ,car tl)
           (prim rest ,cdr tl)
           ,@(make 'if-cont '(k rest env prog) 'cl)
           (lift cl stack)
           (jump expr cl M e0 env prog)))

(const quote quote)
(prim quote? ,eq? quote hd)
(if quote? ((prim e0 ,car tl)
            (lift e0)
            ,@(call 'k '(M e0))))

(const and and)
(prim and? ,eq? and hd)
(if and? ((prim e0 ,car tl)
          (prim rest ,cdr tl)
          ,@(make 'and-cont '(k rest env prog) 'cl)
          (lift cl stack)
          (jump expr cl M e0 env prog)))

(const or or)
(prim or? ,eq? or hd)
(if or? ((prim e0 ,car tl)
         (prim rest ,cdr tl)
         ,@(make 'or-cont '(k rest env prog) 'cl)
         (lift cl stack)
         (jump expr cl M e0 env prog)))
```

```
(const case case)
(prim case? ,eq? case hd)
(if case? ((prim e0 ,car tl)
           (prim rest ,cdr tl)
           ,@(make 'case-cont '(k rest env prog) 'cl)
           (lift cl stack)
           (jump expr cl M e0 env prog)))

(const begin begin)
(prim begin? ,eq? begin hd)
(if begin? ((prim e0 ,car tl)
            (prim rest ,cdr tl)
            (prim one-more ,null? rest)
            (if one-more ((jump expr k M e0 env prog)))
            ,@(make 'begin-cont '(k rest env prog) 'cl)
            (lift cl stack)
            (jump expr cl M e0 env prog)))

(const lambda lambda)
(prim lambda? ,eq? lambda hd)
(if lambda? ((prim e0 ,car tl)
             (prim rest ,cdr tl)
             (prim body ,car rest)
             ,@(make 'lambda-proc '(e0 body env prog) 'lam)
             (lift lam)
             (lift lam 1 stack)
             ,@(call 'k '(M lam))))

(const destruct-name destruct)
(prim is-destruct ,eq? destruct-name hd)
(if is-destruct ((prim e0 ,car tl)
                 (prim rest ,cdr tl)
                 (prim e1 ,car rest)
                 (prim r2 ,cdr rest)
                 (prim body ,car r2)
                 ,@(make 'ds-cont '(k e0 body env prog) 'cl)
                 (jump expr cl M e1 env prog)))
```

```
      (const let-name let)
      (prim is-let ,eq? let-name hd)
      (if is-let ((prim e0 ,car tl)
                  (prim rest ,cdr tl)
                  (prim body ,car rest)
                  ,@(make 'blcont '(k body env prog) 'cl)
                  (const break-let-clauses ,break-let-clauses)
                  (const nil ())
                  (jump break-let-clauses cl M e0 nil nil)))

      ,@(make 'apply-cont '(k hd env prog) 'cl)
      (lift cl stack)
      (const expr-list expr-list)
      (jump expr-list cl M tl env prog)))

(code expr-list
      (k M exprs env prog)
      ((prim is-null ,null? exprs)
       (if is-null ((const nil ())
                    (lift nil (run ,(lambda (self)
                                      (make-spine-static-bt
                                       (find-code 'expr-list-cont2
                                                  self) 'r))))
                    ,@(call 'k '(M nil))))
       (prim e ,car exprs)
       (prim rest ,cdr exprs)
       (const expr expr)
       ,@(make 'expr-list-cont '(k rest env prog) 'cl1)
       (lift cl1 stack)
       (jump expr cl1 M e env prog)))

(code expr-list-cont
      (self M r)
      (,@(unmake '(k rest env prog) 'self)
       ,@(make 'expr-list-cont2 '(k r) 'cl2)
       (lift cl2 stack)
       (const expr-list expr-list)
       (jump expr-list cl2 M rest env prog)))
```

```
(code expr-list-cont2
      (self M r-tl)
      (,@(unmake '(k r-hd) 'self)
       (prim r ,cons r-hd r-tl)
       ,@(call 'k '(M r))))

(code apply-cont
      (self M args)
      (,@(unmake '(k hd env prog) 'self)
       (prim is-prim? ,procedure? hd)
       (if is-prim? ((prim r ,apply hd args)
                     ,@(call 'k '(M r))))

       (const debug debug)
       (prim is-debug? ,eq? hd debug)
       (if is-debug? ((prim v ,car args)
                      (debug v)
                      ,@(call 'k '(M v))))

       (const lift lift)
       (prim is-lift? ,eq? hd lift)
       (if is-lift? ((prim v ,car args)
                     (prim args1 ,cdr args)
                     (prim more? ,pair? args1)
                     (if more? ((prim bt ,car args1)
                                (lift v 1 bt)
                                ; (lift v 1)
                                ,@(call 'k '(M v))))
                     (lift v 1)
                     ,@(call 'k '(M v))))

       (const set-base set-base)
       (prim is-set-base? ,eq? hd set-base)
       (if is-set-base? ((prim v ,car args)
                         (prim args1 ,cdr args)
                         (prim b ,car args1)
                         (lift v base 1 b)
                         ,@(call 'k '(M v))))
```

```
      (const set-memory set-memory!)
      (prim is-set-mem? ,eq? hd set-memory)
      (if is-set-mem? ((prim new-M ,car args)
                       (const ok ok)
                       (lift ok)
                       ,@(call 'k '(new-M ok))))

      (prim named-prim
            ,(lambda (prog name)
               (cond ((assq name prog)
                      => (lambda (p)
                           (if (and (null? (cddr p))
                                    (procedure? (cadr p)))
                               (cadr p)
                               #f)))
                     (else #f)))
            prog hd)
      (if named-prim ((prim r ,apply named-prim args)
                      ,@(call 'k '(M r))))

      (prim proc-call? ,assq hd prog)
      (if proc-call? ((lift k)
                      (lift k 1 stack)
                      (const $inline #f)
                      (const apply apply)
                      (jump apply k M prog hd args)))

      ,@(make 'lambda-call-cont '(k args prog) 'cl)
      (lift cl stack)
      (const expr expr)
      (jump expr cl M hd env prog)))

(code lambda-call-cont
      (self M fn)
      (,@(unmake '(k args prog) 'self)
       (lift k)
       (lift k 1 stack)
       ,@(call 'fn '(k M args))))
```

```
(code if-cont
      (self M p)
      (,@(unmake '(k e12 env prog) 'self)
       (if p ((prim e ,car e12)
              (const expr expr)
              (const $inline #t)
              (jump expr k M e env prog)))
       (prim e2 ,cdr e12)
       (prim e ,car e2)
       (const $inline #t)
       (const expr expr)
       (jump expr k M e env prog)))

(code and-cont
      (self M p)
      (,@(unmake '(k e12 env prog) 'self)
       (if p ((prim e ,car e12)
              (const expr expr)
              (const $inline #t)
              (jump expr k M e env prog)))
       (const false #f)
       (const $inline #t)
       (lift false)
       ,@(call 'k '(M false))))

(code or-cont
      (self M p)
      (,@(unmake '(k e12 env prog) 'self)
       (if p ((const $inline #t)
              ,@(call 'k '(M p))))
       (const $inline #t)
       (prim e ,car e12)
       (const expr expr)
       (jump expr k M e env prog)))

(code case-cont
      (self M key)
      (,@(unmake '(k rest env prog) 'self)
       (const case-loop case-loop)
       (jump case-loop k M key rest env prog)))
```

```
(code case-loop
      (k M key clauses env prog)
      ((prim e? ,null? clauses)
       (if e? ((const error error)
               (lift error)
               ,@(call 'k '(M error))))
       (prim case-e ,car clauses)
       (prim pat-e ,car case-e)
       (prim tl0 ,cdr case-e)
       (prim consq-e ,car tl0)
       (prim tl ,cdr clauses)
       (prim d? ,eq? pat-e key)
       (if d? ((const expr expr)
               (const $inline #t)
               (jump expr k M consq-e env prog)))
       (const $inline #t)
       (const case-loop case-loop)
       (jump case-loop k M key tl env prog)))

; make this a tail call for the last one!!! XXX
(code begin-cont
      (self M r)
      (,@(unmake '(k rest env prog) 'self)
       (prim done ,null? rest)
       (if done (,@(call 'k '(M r))))
       (prim e ,car rest)
       (prim r2 ,cdr rest)
       (prim one-more ,null? r2)
       (const expr expr)
       (if one-more ((jump expr k M e env prog)))
       ,@(make 'begin-cont '(k r2 env prog) 'cl)
       (jump expr cl M e env prog)))

(code blcont
      (self M vars exprs)
      (,@(unmake '(k body env prog) 'self)
       ,@(make 'let-cont '(k vars body env prog) 'cl)
       (const expr-list expr-list)
       (jump expr-list cl M exprs env prog)))
```

```
(code let-cont
      (self M vals)
      (,@(unmake '(k vars body env prog) 'self)
       (const subst subst)
       (jump subst k M vars vals body env prog)))

(code ds-cont
      (self M vals)
      (,@(unmake '(k pat body env prog) 'self)
       (const subst subst)
       (jump subst k M pat vals body env prog)))

(code subst
      (k M vars vals body env prog)
      (,@(make 'subst-cont '(k M body env prog) 'cl)
       (const zip ,root-zip)
       (const nil ())
       (lift nil (run ,(lambda (self)
                         (make-env-bt
                          (find-code 'zip root-zip)))))
       (jump zip cl vars vals nil)))

(code subst-cont
      (self frame)
      (,@(unmake '(k M body env prog) 'self)
       (prim over-spine ,cons frame env)
       (const expr expr)
       (jump expr k M body over-spine prog)))

(code lambda-proc
      (self k M args)
      (,@(unmake '(formals body env prog) 'self)
       (const subst subst)
       (jump subst k M formals args body env prog)))

(define debug-sal
  (code-rec1
   `((code debug
           (self k M v)
           ((debug v)
            ,@(call 'k '(M v)))))))
```

```
(define debug-cl (cons debug-sal 'coccyx))

(define sal-lib
  (append `((set-cache! (cache)
                        (set-memory! (cons (car (get-memory))
                                               cache)))
            (set-space! (space)
                        (set-memory! (cons space
                                               (cdr (get-memory)))))

            (trick (d max) (trick-loop 0 d max))
            (trick-loop (s d max)
                        (if (= s d) s
                            (trick-loop (+ s 1) d max)))

            (even? (i) (zero? (bit-and i 1)))

            (length (l)
                    (if (pair? l)
                        (+ 1 (length (cdr l)))
                        0))

            (compose (f g) (lambda (x) (f (g x)))))

          (map (lambda (op) (list op (eval-at-top op)))
               `(+ = * - imod idiv > >= < <= early= error
                   cons cons3 car cdr null? pair?
                   zero? eq? lax-zero?
                   bit-shift-left bit-shift-right
                   bit-and bit-or bit-not
                   load-word store-word))))
```

A.6 SAL residuals

A.6.1 Factorial

Code generated demonstrating ordinary recursion. Note the duplication, see Section 4.2.5.

```
(fact (x) (if (= 0 x) 1 (* x (fact (+ x -1)))))
→
```

```
((code fact
       (k M x)
       ((const k0 0)
        (prim p1 = k0 x)
        (if p1
            ((const k1 1)
             (prim p car k)
             (jump p k M k1)))
        (const k1 1)
        (prim x-1 - x k1)
        (const pq (code fin ...))
        (const c0 ())
        (prim c1 cons x c0)
        (prim c2 cons k c1)
        (prim k2 closure-cons pq c2)
        (lift k2 0 stack)
        (const p (code fact2 ...))
        (jump p k2 M x-1))
       ...)

 (code fact2
       (k M x)
       ((const k0 0)
        (prim p1 = k0 x)
        (if p1
            ((const k1 1)
             (prim p car k)
             (jump p k M k1)))
        (const k1 1)
        (prim x-1 - x k1)
        (const pq (code fin ...))
        (const c0 ())
        (prim c1 cons x c0)
        (prim c2 cons k c1)
        (prim k2 closure-cons pq c2)
        (lift k2 0 stack)
        (const p (code fact2 ...))
        (jump p k2 M x-1))
       ...)
```

```
 (code fin
       (self M r)
       ((prim d0 cdr self)
        (prim k car d0)
        (prim d1 cdr d0)
        (prim x car d1)
        (prim d2 cdr d1)
        (prim xr * x r)
        (prim p car k)
        (jump p k M xr))
       ...))
```

A.6.2 Even

Code generated demonstrating tail-recursion:

```
(even (x) (if (= 0 x) #t (odd  (- x 1))))
(odd  (x) (if (= 0 x) #f (even (- x 1))))
→
((code even
       (k M x)
       ((const k0 0)
        (prim p1 = k0 x)
        (if p1
            ((const true #t)
             (prim p car k)
             (jump p k M true)))
        (const k1 1)
        (prim x-1 - x k1)
        (lift k 0 stack)
        (const p (code odd ...))
        (jump p k M x-1))
       ...)
```

```
(code odd
      (k M x)
      ((const k0 0)
       (prim p1 = k0 x)
       (if p1
           ((const false #f)
            (prim p car k)
            (jump p k M false)))
       (const k1 1)
       (prim x-1 - x k1)
       (lift k 0 stack)
       (const p (code even2 ...))
       (jump p k M x-1))
      ...)

(code even2
      (k M x)
      ((const k0 0)
       (prim p1 = k0 x)
       (if p1
           ((const true #t)
            (prim p car k)
            (jump p k M true)))
       (const k1 1)
       (prim x-1 - x k1)
       (lift k 0 stack)
       (const p (code odd ...))
       (jump p k M x-1))
      ...))
```

A.6.3 Lambda

Code generated for

```
(object-message (x) ((make-obj x 5) 'vie))
```

```
(make-obj (a b)
          (lambda (msg)
            (case msg
              (urk (* a 2))
              (vie (+ b 3)))))
```

$\rightarrow$

```
((code object-message
       (k M x)
       ((const vie vie)
        (const k5 5)
        (const pq (code fin ...))
        (const c0 ())
        (prim c1 cons vie c0)
        (prim c2 cons k c1)
        (prim k2 closure-cons pq c2)
        (lift k2 0 stack)
        (const p (code make-obj ...))
        (jump p k2 M x k5))
       ...)

 (code make-obj
       (k M a b)
       ((const pq (code unnamed-lambda ...))
        (const c0 ())
        (prim c1 cons a c0)
        (prim c2 cons b c1)
        (prim obj closure-cons pq c2)
        (lift obj 0 stack)
        (prim p car k)
        (jump p k M obj))
       ...)
```

```
(code fin
      (self M r)
      ((prim d0 cdr self)
       (prim k car d0)
       (prim d1 cdr d0)
       (prim msg car d1)
       (prim d2 cdr d1)
       (lift k 0 stack)
       (prim p car r)
       (const c0 ())
       (prim c1 cons msg c0)
       (jump p r k M c1))
      ...)
```

```
(code unnamed-lambda
      (self k M args)
      ((prim d0 identity self)
       (prim lam car d0)
       (prim d1 cdr d0)
       (prim b car d1)
       (prim d2 cdr d1)
       (prim a car d2)
       (prim d3 cdr d2)
       (prim msg car args)
       (prim a2 cdr args)
       (const urk urk)
       (prim p1 eq? urk msg)
       (if p1
           ((const k2 2)
            (prim r * a k2)
            (prim p car k)
            (jump p k M r)))
       (const vie vie)
       (prim p2 eq? vie msg)
       (if p2
           ((const k3 3)
            (prim r + b k3)
            (prim p car k)
            (jump p k M r)))
       (const error error)
       (prim p car k)
       (jump p k M error))
      ...))
```

A.7 Cyclic Integers in Similix

Similix 5.0 is a freely available and widely used specializer and compiler generator. It performs monovariant binding-time analysis, so we must use continuation-passing style for zero and equality testing. Cyclic integers are modeled with partially-static data-structures. Sharing information (and thus a cache) is not supported. Until I find a way to prevent `_sim-memoize` from lifting its argument, dynamic conditionals inside of loops are impossible (note commented out definition of `next` for `append-signal` below.

The `ba.adt` file (this uses SCM's bit operations):

```
(defconstr (cyclic * * *))

(defconstr
  (memory-signal * * * *)
  (constant-signal *)
  (delay-signal * *)
  (map-signal * *)
  (prefix-signal * *)
  (append-signal * * *)
  (prefix-list-signal * * *)
  (binop-signal * * *))

(defprim (divide x y) (inexact->exact (floor (/ x y))))
(defprim 2 << ash)
(defprim (>> x y) (ash x (- y)))
(defprim 2 & logand)
(defprim 2 | logior)
(defprim-dynamic (load-word x) x)
(defprim-dynamic (lift x) x)
(defprim (debug x) (format #t "debug ~S~%" x) x)
```

The `ba.sim` file:

```
(define (push d n)
  (if (zero? n)
      (_sim-error 'push "push out of range")
      (let ((next (- n 1)))
        (if (= d next)
            next
            (push d next)))))

(define (D->C d b)
  (let ((r (modulo d b))
        (q (divide d b)))
    (cyclic b q (push r b))))
```

```
(define (set-base c b^)
  (caseconstr c
    ((cyclic b q r)
     (if (= b b^) c
         (let ((j (lcm b b^)))
           (if (= b^ j)
               (let ((e (divide b^ b)))
                 (cyclic b^
                         (divide q e)
                         (+ (* b (push (modulo q e) e))
                            r)))
               (let* ((e (divide j b^))
                      (b^^ (divide b e)))
                 (set-base (cyclic b^^
                                   (+ (divide r b^^)
                                      (lift (* q e)))
                                   (modulo r b^^))
                           b^))))))))

(define (+c c s)
  (caseconstr c
    ((cyclic b q r)
     (let ((r1 (modulo (+ r s) b))
           (q1 (divide (+ r s) b)))
       (cyclic b (+ q q1) r1)))))

(define (zero?c c t f)
  (caseconstr c
    ((cyclic b q r)
     (if (zero? (modulo r b))
         (_sim-memoize (if (zero? q) (t) (f)))
         (f)))))

(define (count c r)
  (zero?c c
          (lambda () r)
          (lambda () (count (+c c -1)
                            (+ r 7)))))
```

```
(define (nested-count c r)
  (zero?c c
          (lambda () r)
          (lambda () (nested-count (+c c -1)
                                   (+ r (count c (lift 0)))))))

(define (=c c0 c1 t f)
  (caseconstr c0
    ((cyclic b0 q0 r0)
     (caseconstr c1
       ((cyclic b1 q1 r1)
        (if (= b0 b1)
            (if (= r0 r1)
                (_sim-memoize (if (= q0 q1) (t) (f)))
                (f))
            (_sim-error '=c "bases differ: ~S ~S" b0 b1)))))))

(define (/c c s)
  (caseconstr c
    ((cyclic b q r)
     (if (zero? (modulo b s))
         (cyclic (quotient b s) q (quotient r s))
         (_sim-error '/c "uneven ~S ~S" b s)))))

(define (%c c s)
  (caseconstr c
    ((cyclic b q r)
     (if (= b s) r
         (_sim-error '%c "uneven ~S ~S" b s)))))

(define (mask b) (- (<< 1 b) 1))
```

```
(define (load-sample p b)
  (let* ((W 32)
         (wa (/c p W))
         (ba (%c p W))
         (w0 (load-word wa)))
    (if (<= (+ b ba) W)
        (& (mask b) (>> w0 ba))
        (let* ((under-by (- W ba))
               (s0 (& (mask under-by) (>> w0 ba)))
               (w1 (load-word (/c (+c p under-by) W)))
               (s1 (& w1 (mask (- b under-by)))))
          (| s0 (<< s1 under-by))))))

(define (sum start stop size stride rez)
  (=c start stop
      (lambda () rez)
      (lambda () (sum (+c start stride) stop size stride
                      (+ rez (load-sample start size))))))

(define (get s)
  (caseconstr s
   ((memory-signal start stop size stride)
    (load-sample start size))
   ((constant-signal c) c)
   ((delay-signal v s) v)
   ((prefix-signal v s) v)
   ((prefix-list-signal hd tl s) hd)
   ((append-signal hd tl s1) hd)
   ((map-signal f s) (f (get s)))
   ((binop-signal f s0 s1)
    (f (get s0) (get s1)))))
```

```
(define (end? s t f)
  (caseconstr s
   ((memory-signal start stop size stride)
    (=c start stop t f))
   ((constant-signal c) (t))
   ((delay-signal v s) (end? s t f))
   ((prefix-signal v s) (f))
   ((prefix-list-signal hd tl s) (f))
   ((append-signal hd tl s1) (f))
   ((map-signal op s) (end? s t f))
   ((binop-signal op s0 s1)
    (end? s0 (lambda () (end? s1 t f)) f)))) ; duplication

(define (next s)
  (caseconstr s
   ((memory-signal start stop size stride)
    (memory-signal (+c start stride) stop size stride))
   ((constant-signal c) s)
   ((delay-signal v s) (delay-signal (get s) (next s)))
   ((prefix-signal v s) s)
   ((prefix-list-signal hd tl s)
    (if (null? tl) s (prefix-list-signal (car tl) (cdr tl) s)))
   ((map-signal f s) (map-signal f (next s)))
#| ((append-signal hd tl s1)
    (end? tl
          (lambda () s1)
          (lambda ()
            (append-signal (get tl) (next tl) s1)))) |#
   ((binop-signal f s0 s1)
    (binop-signal f (next s0) (next s1)))))

(define (plus x y) (+ x y))
(define (times x y) (* x y))

(define (reduce s r f)
  (end? s
        (lambda () r)
        (lambda () (reduce (next s) (f r (get s)) f))))
```

```
(define (filter prefix kernel in)
  (if (null? prefix)
      (constant-signal 0)
      (binop-signal plus
                    (map-signal (lambda (v)
                                   (* (car kernel) v))
                                in)
                    (filter (cdr prefix)
                            (cdr kernel)
                            (delay-signal (car prefix)
                                          in)))))
```

A.8 Delimited Control: shift/reset

The material in this section is extracted from Sections 2 and 5.2 of [DaFi92]. Shift is similar to Scheme's `call/cc` (call with current continuation), but the extent of the escape procedure is limited by reset. Figure A.1 gives the formal semantics of shift and reset. Conceptually, they serve as composition and identity of continuations. [LaDa94] provides an explanation their application to partial evaluation.

Here is a simple example:

1 + reset (2 * shift c in 3 * ((c 4) + (c 5))) $\rightarrow$
1 + (let c = λ v . 2 * v in 3 * ((c 4) + (c 5))) $\rightarrow$
55

In the following example, ϕ is a boolean-valued term with free variables b_i. `Sat` is true if and only if ϕ is satisfiable. Each call to the `flip` function tries returning twice, so every possible assignement of truth values is tried until ϕ is satisfied.

fun flip () = shift c in ((c true) orelse (c false))

val sat = reset let b_0 = flip ()
b_1 = flip ()
...
in ϕ

$$
\begin{aligned}
[\![x]\!] &= \lambda\kappa.\kappa x \\
[\![\lambda x.M]\!] &= \lambda\kappa.\kappa\ (\lambda x.\ [\![M]\!]) \\
[\![MN]\!] &= \lambda\kappa.\ [\![M]\!]\ (\lambda m.\ [\![N]\!]\ (\lambda n.mn\kappa)) \\
\\
[\![\text{shift } c \text{ in } M]\!] &= \lambda\kappa.\ \text{let } c = \lambda a.\lambda\kappa'.\kappa'(\kappa a) \text{ in } [\![M]\!]\ (\lambda m.m) \\
[\![\text{reset } M]\!] &= \lambda\kappa.\kappa\ ([\![M]\!]\ (\lambda m.m))
\end{aligned}
$$

Figure A.1: The first three lines are the standard equational specification of translation of λ_v terms into continuation-passing style (CPS). The last two lines define shift and reset.