Compiler Generation for Interactive Graphics using Intermediate Code

Scott Draves

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue, Pitsburgh, PA 15213, USA
Home page: `http://www.cs.cmu.edu/~spot`

Abstract. This paper describes a compiler generator (cogen) designed for interactive graphics, and presents preliminary results of its application to pixel-level code. The cogen accepts and produces a reflective intermediate code in continuation-passing, closure-passing style. This allows low overhead run-time code generation as well as multi-stage compiler generation. We extend partial evaluation techniques by allowing *partially static integers*, conservative early equality, and unrestricted lifting. In addition to some standard examples, we examine graphics kernels such as one-dimensional finite filtering and packed pixel access.

1 Introduction

Interactive graphics is a growing application domain where the demands of latency, bandwidth, and software engineering collide. The state of the art, represented by systems such as QuickDraw GX(R) [44], Photoshop(tm) [42], RenderMan(tm) [52], Explorer(R) [25], and DOOM(tm) [10], is to write in C and assembly language. Programmers use hand-specialized routines, buffering, collection-oriented languages [46], and embedded/dynamic languages, but inevitably we face trade-offs in

program size Large libraries with many specialized but infrequently used routines waste space.

latency and memory Using larger batches/buffers reduces interpreter overhead but increases latency and memory traffic.

design time Optimization requires time and planning that are unavailable to exploratory and evolutionary programmers.

Run-time code generation (RTCG), as exemplified by Common LISP [51], Pike's Blit terminal [43], Masselin's Synthesis operating system [36], researchers at the University of Washington [32], and elsewhere [14][35] is one way to attack this problem. With these systems, one writes programs that create programs. Generating rare cases and fused, one-pass loops as needed directly addresses the program size and latency trace-offs outlined above.

However, RTCG has suffered from a lack of portable, easy-to-use interfaces. Lisp's quasi-quote, Scheme's `syntax-rules` [5], and parser generators such as YACC [27] automate the mechanics of constructing certain classes of programs, but it remains unclear how we can build an RTCG system that is effective on a wide-range of problems, and is automatic enough that design time and programmer effort can really be reduced.

Partial evaluation (PE) as described in [28] is a semantics-based program transformation. With the cogen approach the programmer can type-check and debug a one-stage interpreter, then by annotation and tweeking, produce an efficient two-stage procedure (a compiler). Binding times manage program division, memoization handles circularities, and the specializer creates variable names and the rest of the mechanics of code construction. The programmer concentrates on higher-level issues such as staging and generalization. Other current attempts to apply PE to RTCG are Fabius [35] and Tempo [8].

This paper explores the application of a directly implemented compiler generator for an intermediate language to pixel-level graphics kernels. The nature of graphics loops is exploited with cyclic integers, which make the remainder (eg modulo 32) of an integer static. A conservative static-equality-of-dynamic-values operator allows static elimination of software caches, thus reducing memory references. The combination of these features allows us to convert bit-level code to word-level code.

For example, say one were converting a packed 24-bit RGB image to 8-bit grayscale. An efficient implementation reads three whole words, breaks them into twelve samples with static shifts and masks, computes the four output bytes, and assembles and writes an output word (see Figure 1) Such a block can make good use of instruction level parallelism. Our objective is to produce residual code like this from a general routine (called say `image-op`) that can handle any channel organization, bits per pixel, per-pixel procedure, etc.

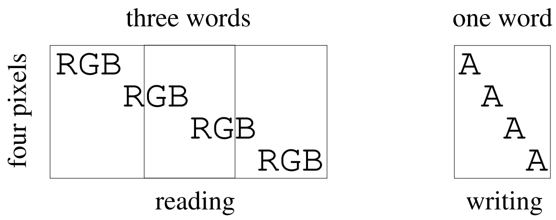

Fig. 1. efficient pixel access

The rest of this paper consists of a system overview followed by a description of `cogen`, two examples, and experimental results. Sections 6 and 7 place the system in context and conclude. Readers not interested in the intermediate language and its effect on cogen might read only Sections 2, 4.1, and 4.2.2 before skipping to the examples in Section 5.

More discussion but few details can be found in [11]. This paper assumes the reader is familiar with binding times, C compilers, LISP macros, caches, and pixels. [16] provides a good introduction to graphics and [22] to chip architecture.

2 System Overview

Our system is called *Nitrous*; within it, we identify three kinds of program transformers (see Figure 1):

- front ends (traditional and generated) which produce `root` programs from programs in user-defined languages.
- a compiler generator `cogen` for an untyped intermediate language `root`.
- a backend for code assembly and the rest of the run-time system.

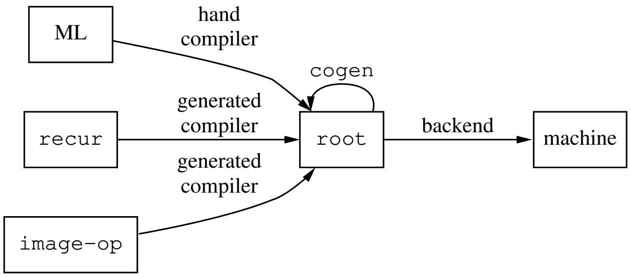

Fig. 2. System Diagram

`root` is a simple abstract machine code, like quad-code [1] with an unlimited number of registers but in continuation-passing closure-passing style (CPS-CPS) [3]. Thus the stack and closures are explicit data structures and all values are named uniformly. The model includes reflection and reification [17], simple data structures, arithmetic, an open set of primitive functions, and represents higher-order values with closures.

`recur` is a sample front end. It is a simple recursive equations language with parallel `let`, `if`, and multi-argument procedures. The `cogen`-created compiler produces straightforward code. It compiles tail-recursive calls without building stack frames, but is otherwise non-optimizing.

`Image-op` is the hypothetical procedure described above. The compiler might be run when the user opens a new file, repositions a window on the screen, chooses a new brush, etc.

A compiler generator transforms an interpreter into a compiler. That is, `cogen` transforms a `root` program and *binding times* (BTs) for its arguments into a *generating extension*. The BTs categorize each argument as static program or dynamic data; essentially they are *types* [21]. The extension consists of a memo table, followed by the static parts of the computation interleaved with instructions that generate residual code (ie do RTCG).

The backend executes `root` programs. We examine current and future backends in Section 5.3.

By supporting reflection we make code-producing functions first class. Nitrous takes this a step further by making the compiler-producing function (`cogen`) first class: rather than working with files, it just maps procedures to procedures. To facilitate this the back-end supports reification: the `root` text of any code pointer can be recovered.

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`. The lift compiler (see Section 4.2) works this way; other possibilities include using `recur` to create input for `cogen`, or providing a compiler generator as a primitive in the `recur` language. Such multi-stage application requires that the generated compilers create correctly annotated programs, which can be difficult. In [18] and [19] Glück and Jørgensen present more rigourous and automatic treatments of layered systems using specializer projections and multi-stage binding-time analysis.

3 The Intermediate Language

The core of the system is the intermediate language `root`. Its formal syntax appears in Figure 3. 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 streams. Streams always terminate with a `jump` instruction, which transfers control to the `code` bound to the first argument and passes the rest. Formal semantics can be found in [11].

code	$\longrightarrow$	`(code` *name args instrs*`)`	*v*	$\longrightarrow$	*variable*
instr	$\longrightarrow$	`(prim` *v prim* `.` *args*`)`	*instrs*	$\longrightarrow$	*instr* `list`
	|	`(const` *v constant*`)`	*args*	$\longrightarrow$	*variable* `list`
	|	`(if` *v true-branch*`)`	*true-branch*	$\longrightarrow$	*instrs*
	|	`(jump` *v* `.` *args*`)`	*prim*	$\longrightarrow$	*primitive operation*

Fig. 3. `root` syntax

Structured higher-order control flow is managed with closure-passing [3]. A closure pairs a code pointer with its bound variables, and is invoked by jumping to its car and passing itself as the first argument. 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 for an example.

Factors that weight in favor of an intermediate language like `root` include: `root` 1) makes `cogen` smaller and easier to write; 2) provides target for a range of source languages; 3) provides an interface for portability; 4) opens opportunity to schedule large blocks and utilize instruction level parallelism; 5) exposes language mechanism (such as complex optional arguments and method lookup) to partial evaluation; 6) reduces as-

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

Fig. 4. `root` code for `append`, in sugary-syntax. Notes: *a* return by jumping to the car of `k`, passing `k` and `m` as arguments. *b* `close` is like `cons`, but identifies a closure. *c* destructuring assignment.

sembly overhead because it is essentially an abstract RISC code; 7) allows expressing optimizations not possible in a High Level Language.

And against: 1) types would simplify the implementation and formalization; 2) good loops (eg PC-relative addressing) are difficult to produce; 3) explicit stacks and exceptions would reduce consistency requirements and make optimization easier; 4) using a language like GCC [49], OmniVM [48], or the G-machine [29] would leverage existing research.

4 The Compiler Generator

`cogen` is directly implemented (rather than produced by self-application), polyvariant (allows multiple binding time patterns per source procedure), handles higher-order control flow, and is based on abstract interpretation. This section summarizes how `cogen` and its extensions work, in theory and practice. The subsections cover binding times, cyclic integers, lifting, termination, and special primitive functions in greater detail.

`cogen` converts a `code` and a binding-time pattern to an extension. An extension is identified with the name of the code pointer and the BT pattern, for example `append(D S D)`.

The extensions memoize on static values to produce programs with loops. Arbitrary dynamic control flow can be produced: a recursive equations language can specify any graph. The interaction between cyclic arithmetic and memoization can result in a least common multiple (LCM) computation.

To support variable splitting and inlining the extension renames the variables in the residual code and keeps track of the *shapes* (names and cons structure) of the dynamic values. The shape of a dynamic value is the name of its location. Shapes are part of the key in the static memo table; two shapes match 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. The effect of variable splitting is that the members of a structure can be kept in several registers, instead of allocated on the heap (abstracted into one register).

Inlining is controlled by the dynamic conditional heuristic [4], but setting the special `$inline` variable overrides the heuristic at the next jump.

In CPS-CPS continuations appear as arguments, so static contexts are naturally propagated. Figure 5 shows the translation of `(+ S (if D 2 3))` into `root`. The extension `dyn-if(D S D)` calls the extension `cont((`$\widetilde{close}$ `S (`$\widetilde{list}$ `D S D)) D)`.

```
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)
}
```

Fig. 5. Propagating a static context past a dynamic conditional.

4.1 Binding Times

Binding times are the *metastatic* values from self-applicable PE. They represent properties derived from the interpreter text while a compiler generator runs. 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.

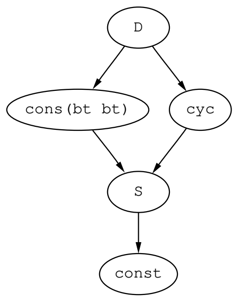

Fig. 6. Lattice Order. $(\widetilde{const}\ c) \sqsubset \mathtt{S} \sqsubset \mathtt{cyc} \sqsubset \mathtt{D}$

`cogen`'s binding-time lattice appears in Figure 6. Cons cells are handled with graph grammars as in Mogensen [37]: pairs in binding times are labeled with a 'cons point'. If the same label appears on a pair and a descendant of that pair then the graph is collapsed, perhaps forming a circularity. An annotation may provide the label, much like a type declaration.

We denote a pair $(\widetilde{cons}\ \ bt\ \ bt)$ (the label is invisible here). $(\widetilde{list}\ \ x\ \ y\ \ \ldots)$ abbreviates $(\widetilde{cons}\ \ x\ \ (\widetilde{cons}\ \ y\ \ \ldots\ \ (\widetilde{const}\ \ \texttt{nil})))$. In the lattice, $\texttt{S} \sqsubset (\widetilde{cons}\ \ bt\ \ bt) \sqsubset \texttt{D}$. We use familiar type constructors to denote circular binding times. Figure 7 depicts several useful examples.

For example, a value with BT `D list` has no static value, but its shape is a list of variable names. The dynamic values are placed in registers as space permits.

As in Schism [6], 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 BT is denoted $(\widetilde{const}\ \ c)$, or just c.

Closures are differentiated from ordinary pairs in the `root` text, and this distinction is maintained in the 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. It is not denoted or discussed further.

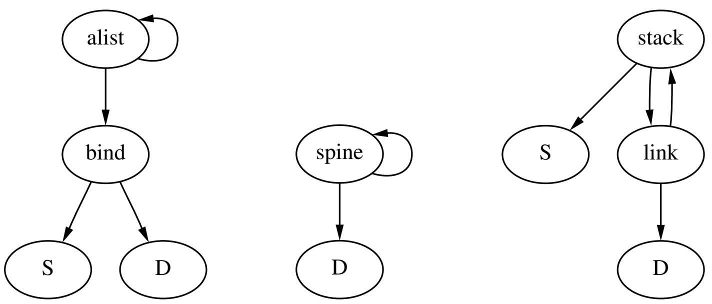

Fig. 7. Three binding times: `(S * D) list` is an association list from static keys to dynamic values, `D list` is a list only whose length is static, *stack-bt* is the binding time of a control stack.

Cyclic Integers There are many ways of dividing integers. Nitrous can break an integer into static base, dynamic quotient, and static remainder[1]: $i = bq+r$. Such a binding time is denoted `cyc`. If b is a power of two (eg 32) then we have a static bit field (eg the 5 low bits are static). In the lattice, $\texttt{S} \sqsubset \texttt{cyc} \sqsubset \texttt{D}$.

We have to make special cases of all the primitives that handle cyclic values. The easiest are addition and multiplication. The static code works like this:

```
v = (+ s (cyclic b q r)) → v = (cyclic b q (+ s r))
v = (* s (cyclic b q r)) → v = (cyclic (* s b) q (* s r))
v = (+ s d) → v = (cyclic 1 d s)
v = (* s d) → v = (cyclic s d 0)
```

[1] Here's another way to divide integers: a static bitmask divides the bits into static and dynamic.

On the left is a source instruction with binding times, on the right is the code in the extension. No case has any dynamic component; because the quotient passes through unchanged we can just copy the shape (`q`). Rules for `(+ cyc cyc)` and `(* cyc cyc)` might also be useful, but are not explored in this paper.

`zero?`, `imod`, and `idiv` are more compicated because the binding time of the result depends on the static value. For `zero?`, if the remainder is non-zero, then we 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 binding times. It makes direct use of polyvariance.

```
v = (zero? (cyclic b q r)) → if (zero? (imod r b))
                                emit v = (zero? q)
                                v = #f
```

See Section 4.2.2 for a brief description of the other affected primitives.

Note that the rule given for addition doesn't constrain the remainder to $0 \leq r < b$ by overflowing into the quotient as one would expect. Instead, the congruence modulo b is maintained only at memo points: this is *late normalization* of cyclic values. The extra information propagated by this technique (early-equality across cycles) is required to handle multiple overlapping streams of data.

Thus when the compiler begins to make a dynamic code block, all cyclic values are normalized by adjusting r to satisfy $0 \leq r < b$. This is done by emitting counter-acting additions to q. The sharing between these values must be maintained across this adjustment.

4.2 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. 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. It is replaced with its extension, this requires a binding time.
- 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.
- a label repeats in a binding-time grammar, causing its collapse.
- a lift directive appears, generally the result of manual annotation.

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

1. `S` → `D`   allocates a dynamic location and initializes it to the static value.
2. `cyc` → `D`   emits a multiplication by the base (unless it is one) and addition with the remainder (unless it is zero).
3. `S` → `cyc`   results from an annotation used to introduce a cyclic value. The conversion is underconstrained; currently a base of one is assumed.
4. ($\widetilde{cons}$ `D D`) → `D`   emits a dynamic `cons` instruction.

5. ($\widetilde{\mathit{close}}$ ($\widetilde{\mathit{const}}$ *p*) *frame*) $\rightarrow$ `D` generates and inserts a call to *p* (($\widetilde{\mathit{close}}$ `D` *x*) `D D ...`) (all but the first argument are `D`), then emits the cons.
6. ($\widetilde{\mathit{close}}$ `S` *frame*) $\rightarrow$ `D` same as the previous case, but the extension has already been computed, so just emits a call and a cons.

Case 5 is 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 4.3 below. The lift compiler handles applying these cases to structured and circular binding times.

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

instruction $\longrightarrow$ `...` | `(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 rudimentary lift language.

Lifting Structures If a lift isn't 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.

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).

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.

Special Primitive Functions `cogen` treats some primitive functions specially, generally in order to preserve partially static information. Figure 8 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 summed with an atom. `null?` and `atom?` are also supported.

e `apply` takes two arguments: a primitive (in reality, a C function) 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 doesn't improve the binding times, it just generates better code.

f extensions for both `S` and `D` are created, the compiler chooses one statically (see Section 4.1.1).

g the source instruction on the left produces the static code on the right. Rather than the error, one could take a dynamic path, as does case *e* above.

```
v = (imod (cyclic b q r) s) → if (zero? (imod b s))
                                v = (imod r b)
                                error
```

h like `imod` but

```
v = (idiv (cyclic b q r) s) → if (zero? (imod b s))
                                v = (cyclic (idiv b s) q r)
                                error
```

Note that it is necessary that the division primitive round down even for negative inputs, ie `(idiv -1 10) → -10`.

i `early=` conservative static equality of dynamic values, see Section 5.1.

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

Fig. 8. See the text for notes.

4.3 Details and Complications

Static Control Stacks 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`)` `= 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. However, to simplify the implementation `cogen` uses a special lift that supplies the return BT(s) and sets up the stack:

instruction $\longrightarrow$ `...` | `(lift` *v* `stack` *ret-bt* `...)`

This causes the lift `(`$\widetilde{close}$ *p* *frame*`)` $\rightarrow$ *stack-bt* and uses extension *p*`((`$\widetilde{close}$ `S` *frame*`)` *ret-bt* `...)` to create the extension for the continuation.

For example, consider `append(D S D)`[2], with lift directives as it shown in Figure 8. The key sequence of extensions and lifts that create the recursion appears in Figure 9. `cogen` could avoid producing the (probably over-) specialized entry/exit code by checking for the end of the stack explicitly.

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

cont(self r) {
  (k l m) = (cdr self)
  nr = (cons (car l) r)
  lift nr
  (car k)(k nr)
}
```

Fig. 9. Annotated code for a static recursion.

```
append(D S D)
(close cont (list D S D))
      → stack-bt
cont((close S (list D S D)) D)

append(stack-bt S D)
cont((close S (list
    stack-bt D S)) D)
```

Fig. 10. Building a static recursion.

Shape un/lifting and Sharing Here we consider lift case 5 from Section 4.2 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.

[2] Only a more complex recursion really requires this, but `append` is easier to understand.

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 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 `recur` program from the stack in the `recur` interpreter? Say `cogen` encounters `do-`$\widetilde{\texttt{call}}$`(`*stack-bt* `S S` *alist-bt*`)` (see Figure 10). When `cl` is lifted we compute `cont((`$\widetilde{\mathit{close}}$ `S (`$\widetilde{\mathit{list}}$ *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.

```
do-call(k fn exp env) {       cont(self arg) {
  frame = (list k fn)           (k fn) = (cdr self)
  cl = (close cont frame)       lift k
  lift cl stack dynamic         $inline = #f
  eval(cl exp env)   a          apply(k fn arg)
}                             }
```

Fig. 11. annotated code to produce a dynamic stack frame. Notes: *a* the call to evaluate the argument is inlined.

Alternative Representations of Cyclic Values Cyclic values as described in Section 4.1 are inadaquate for the filter example described below. 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. The 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 on how the value is used. A simple constraint system should suffice to pick the correct representation.

5 Experiments

This section presents examples of the code transformations possible with Nitrous, and measures their effect on time consumed. Two graphics examples are examined in the subsections, then the benchmark data is presented.

These examples make novel use of partial evaluation to optimize memory access by statically evaluating alignment and cache computations. Thus code written using a `load-nybble` procedure (with bit pointers) can be converted to code that uses `load-word` and static shifts and masks.

5.1 Sequential Nybble Access

Say we sequentially access the elements of a vector of packed sub-word-sized *nybbles*[3]. Figure 12 gives code for reducing a vector of nybbles. The code on the right is specialized to 8 bits per nybble and a bit vector length to zero mod 32. There are three things going on:

- Because the loop index is cyclic, three `zero?` tests are done in the compiler before it reaches an even word boundary and emits a dynamic test. These adjacent iterations of the loop can run in parallel.
- The shift offset is static because `(imod cyc S)` is static. Shifts with constant offsets generally take one fewer register of space and one fewer cycle in time.
- Redundant loads are eliminated by inserting a static cache: we use the procedure `load-word_c` instead of the `load-word` primitive.

The hard part is making the cache work: the cache-present test has dynamic arguments, but it must be eliminated. This is exactly the purpose of `early=`, it returns true if the compiler can prove the values are equal (are aliases). Since the shapes track the locations of the dynamic values, this is accomplished just by testing them for equality. Note how this equality information is propagated through the `idiv` primitive.

Note that in the actual implementation, a store is threaded through the code to provide the state for the memory and its cache.

Even if the index were completely dynamic we could still use this fast loop by applying the Trick to make it cyclic.

5.2 1D Filtering with Software Cache

A one-dimensional finite-response filter transforms an input stream of samples into an output stream by taking a sliding dot-product with a constant kernel-vector (see Figure 14). If the kernel has length k then each word is loaded k times. If one makes the outer loop index be cyclic base k, then in the residual code the loop is expanded and the loads are shared. A window on memory is kept in registers but rather than rotating it, we rotate the code around it.

As before, this can be done using a caching load procedure, though the cache now must maintain several values. A cache has BT `(cyc * S * D) list`, the tuple is of the address, the dirty bit (or other cache control information), and the word from memory. The length of the list controls the cache's size, this must be set manually. The cache

[3] Historically 'nybble' means precisely four bits. I have adopted the term to mean anything from one to thirty-one bits.

```
i = cyclic; sum = D
while (i) {
  i = i - bpn
  nyb = load-nybble(i bpn)
  sum += nyb
}

load-nybble(i bpn) {
  addr = i / 32;
  offset = i % 32;
  word = load-word_c(addr)
  return w2n(word bpn offset)
}
```

```
while (iq) {
  iq = iq - 1
  w = load_word(iq)
  sum += (w >> 24) & 255
  sum += (w >> 16) & 255
  sum += (w >>  8) & 255
  sum += (w >>  0) & 255
}
```

Fig. 12. General and specialized code to reduce a vector of nybbles. For simplicty, this code doesn't handle nybbles that overlap words.

```
load-word_c(cache)(addr) {
  if (early= addr cache.addr)
     w = cache.word
     w = (load-word addr)
  cache = (list addr w)
  return w
}
```

```
w2n(w bpn ny) {
  mask = ((1 << bpn) - 1)
  r = mask & (w >> (ny * bpn))
  return r
}
```

Fig. 13. Helper functions.

is managed with the Least Recently Used (LRU) policy. The entry code that 'fills the pipeline' is produced automatically because the memo-test doesn't hit until the cache gets warm.

This example was the motivation for the late normalization described above. The address `p` is kept premultiplied to avoid a dynamic multiplication when it is lifted at every load.

5.3 Implementation, Backend, and Benchmarks

`cogen` is written in Scheme48 [30], which compiles an extended Scheme to bytecode[4]. `cogen` is 2000 lines, supported by 4000 lines of utilities, the virutal `root` backend, the compiler to GCC, examples, test cases, etc. It has not yet been optimized for speed (eg it doesn't use hashing or union-find). The source code and transcripts of sample runs are available from `http://www.cs.cmu.edu/ spot/nitrous.html`.

[4] In fact the bytecode interpreter is written in Pre-Scheme [31] and compiled to C.

```
stride = S; kernel = S
p = cyc; q = D; klen = S
while (p) {
  dp = dot(0 p kernel klen)
  store_word(q dp)
  p -= stride
  q--
}

dot(sum i j n) {
  while (n--)
    sum += load-word_c(i++)
         * load-word_c(j++)
  return sum
}
```

```
w0 = load-word(p)
w1 = load-word(p-1)
w2 = load-word(p-2)
sum = 2*w0+5*w1+2*w2
store-word(q sum)
p -= 3; q++
while (p-=3) {
  w0 = load-word(p)
  sum = 2*w1+5*w2+2*w0
  store-word(q sum)
  w1 = load-word(p-1)
  sum = 2*w2+5*w0+2*w1
  store-word(q+1 sum)
  w2 = load-word(p-2)
  sum = 2*w0+5*w1+2*w2
  store-word(q+2, sum)
  q += 3;
}
```

Fig. 14. General and specialized finite filter code. The kernel is `[2 5 2]` and the stride is one.

Except for the lift compiler (see Section 4.2) the only working front end is a macro assembler, using Scheme as the macro language. So far the code fed to cogen has been written in `root` (with lift annotations) by hand. The `recur` compiler produces good code but is unfit for use as a front end because it does not yet produce annotated code.

The hypothetical ideal backend performs register allocation, instruction selection and scheduling, dead-code elimination, constant sharing, and linking to convert this language into executable code. I am budgeting about 2000 instructions to produce each dynamic instruction.

The backend used to produce these benchmarks translates a whole `root` program to a single GCC [49] function which is compiled, run, and timed using ordinary Unix(tm) tools. The run-time is just 350 lines of C and does not support garbage collection. These times are on a 486DX4/75 running linux (x86), and a 150Mhz R4400 SGI Indy (mips).

Support for reification and reflection is trivial in the interpreted virtual machine, but non-existent in the GCC backend. The ideal backend would support reflection either by transparent lazy compilation or with a 'compiling eval' procedure in the run-time. Reification could be supported by keeping a backpointer to the intermediate code inside each code segment (unless proven unnecessary).

The benchmark programs are summarized below:

`recur` the front end described in Section 2 run on three simple programs. `atree` performs a mixture of arithmetic and procedure calls, `even` tests the parity of 100 by mutual recursion, `append` a list of length 5 in the usual way.

nybble reduce a vector of nybbles as above, but handles nybbles that cross word boundaries. Ran on 2500 bytes with nybbles of size 4 and 12.

filter a vector of integers, kernel size 3 and 7.

In the nybble and filter examples, the nitrous-int code uses calls to the cached memory operations, but the cache size is set to zero. The manual-int code uses ordinary loads.

The numbers are reported in Figure 15. Appreciable speed-ups are achieved in most cases. The hand-written C code is about twice as fast as the compiled `root` programs. We speculate that this is because we use 'computed goto' and the `&&` operator for all control flow.

	mips				x86			
	nitrous		manual		nitrous		manual	
	int	spec	int	spec	int	spec	int	spec
recur atree	100	2.5			600	15		
recur even	1900	3.7			11000	13		
recur append	150	3.6			710	23		
nybble 4	12000	160	380	140	64000	460	1500	380
nybble 12	4700	77	180	59	25000	210	680	170
filter 3	6500	140	400	45	53000	620	650	230
filter 7	13000	380	880	68	760	1400	450	

Fig. 15. Benchmark data, times in microseconds, two digits of precision. All time trials run five times; best time taken. The 'int' columns are for the general interpreters; and 'spec' for the specialized residual code. The 'nitrous' columns are for code written in `root` and produced by `cogen`, and 'manual' for normal C code written by hand.

6 Related Work and Alternate Paths

This section places Nitrous in context of computer graphics systems practice, and other research in partial evaluation and RTCG. First, we list the standard approaches to the generality/performance trade-off with a collection of examples of each.

custom hardware Blitters, MPEG codec chips. Provides the highest performance at the highest price with the greatest design time and least flexibility.

programmable hardware DSP chips, MediaProcessor(tm) [38]. Require assembly language and special tools to fully utilize them, but they can run C.

application specific compilers MINT [53], Apply [20], Cellang [13]. Usually compile to C and run on stock hardware.

dynamic linking Photoshop(tm) [42], Netscape(tm) [39]. Known as 'plug-ins'. The application dynamically loads code modules adhering to published interfaces.

batching/buffering APL, RenderMan(tm) [52], fnord [12]. Rather than applying an interpreter and one program to each of many data, a batch interpreter sequences vector primitives over the data, thus reducing interpreter overhead. Strip-mining and tiling [55] is necessary if the data don't fit in the cache.

embedded/dynamic languages Emacs lisp [50], Tcl/Tk [40], Microsoft's Visual Basic(tm) [54], PostScript(tm) [24], ScriptX [45], etc. The dynamic language sequences routines implemented with a static languages.

Hardware support for byte-pointers and an on-chip cache have similar effect as our loop optimizations (the hardware repeats the computation, but the hardware is very fast). However, note that the Alpha [47] doesn't support byte pointers, and DSP chips sometimes provide an addressing mode for on-chip SRAM bank, rather than a cache. This is an application of RISC philosophy (factoring from hardware into the compiler to increase the clock rate).

Similix [4] is a sophisticated, freely-available compiler generator. It uses a type-inference BTA, supports higher order Scheme-like language with datatypes and an open set of prims. It supports partially static structs, simple manual lifts, and is monovariant. It produces small programs and runs fairly quickly. It's file-based interface could be combined with a Scheme compiler (provided it had the right interface) to do RTCG. Schism [7] is similar but nicer.

DCG [14] and [33] provide C-callable libraries for RTCG. They use typical C-compiler intermediate language for portable construction and fast compilation with 'rudimentary optimization'. The ratio of static instructions used per dynamic instruction produced is 300 to 1000.

'C [15] augments C with backquote-like syntax to support manual RTCG. It provides a nice interface to DCG, and can handle complex interpreters (eg Tiny-C). By retargetting C-mix [2] to 'C one might be able to combine the strengths of these systems (constraint-based BTA, fast code generation, a popular language).

Fabius [35] is a compiler generator for a simplified first-order ML-like language. The programmer uses curry notation to specify the program division, and a BTA completes it. This is a very natural means of annotation. The compilers produce machine code directly, thus they are very fast; its ratio is about 7.

Tempo [8] is a off-line, template-based specializer for C aimed at operating systems code. It contains sophisticated pointer analyses and other features to make it work on 'real' systems. So far no results are available.

Staging transformations [26], ordered rewriting [9], program slicing [23], and metaobject protocols [34] contain related ideas from other parts of the language research community.

7 Conclusion and Future Directions

We have described Nitrous, a run-time code generation system for interactive graphics. It uses compiler generation of intermediate code to provide sophisticated transformations with low overhead. It augments standard partial evaluation techniques with new annotations and binding times. While the preliminary results from the graphics kernels are promising, the front and backend are still too incomplete to conclusively demonstrate the utility of this approach. Besides the immediate goals of fleshing out the system and scaling up the experiments, we hope to

- formalize the binding-time lattice in Elf [41] and develop a constraint based analysis.
- merge the static and shape environments, if possible.
- reduce unnecessary memoization.
- reduce the size of the extensions by improving lifting.
- return to self-application.
- bootstrap the system by compiling the experiments (and ultimately itself) with a generated compiler.
- when shape un/lifting data, use vectors instead of lists.
- automatically pick the right representation (pre/post multiplied) of cyclic values.
- can dynamic just be a special case of cyclic (when base is one)?

8 Acknowledgments

This paper was partially written and researched while visiting DIKU and DAIMI with funding from the Danish Research Council's DART project. I would like to thank Olivier Danvy, Nick Thompson, and the anonymous reviewers for their comments on drafts of this paper, and Peter Lee for his continuing feedback, faith, and support.

References

1. A V Aho, R Sethi, J D Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley 1986.
2. Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. DIKU 1994.
3. Andrew Appel. *Compiling with Continuations*. Cambridge University Press 1992.
4. A Bondorf, O Danvy. Automatic Autoprojection of Recursive Equations with Global Variables and Abstract Data Types. *Science of Computer Programming*16:151-195.
5. William Clinger, Jonathan Rees. Revised^4 Report on the Algorithmic Language Scheme. *LISP Pointers*IV:1-55.
6. Charles Consel. Binding Time Analysis for Higher Order Untyped Functional Languages. *ACM Conference on Lisp and Functional Programming*, 1990.
7. Charles Consel. New Insights into Partial Evaluation: The Schism Experiment. *European Symposium on Programming*, 1988.
8. 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.
9. N Dershowitz, U Reddy. Deductive and Inductive Synthesis of Equational Programs. *Journal of Symbolic Computation*15:467-494.
10. DOOM. id Software 1993.
11. Scott Draves. Lightweight Languages for Interactive Graphics. CMU-CS-95-148.
12. Fnord: a Visualization System for Differential Geometry. Brown University 1991.

13. Cellang. ? 1995.
14. Dawson Englar, Todd Proebsting. DCG: An Efficient, Retargetable Dynamic Code Generation System. *ASPLOS*, 1994.
15. 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.
16. Foley, Feiner, Andries van Dam, John Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley 1990.
17. Daniel P Friedman, Mitchell Wand. Reification: Reflection without Metaphysics. *ACM Conference on Lisp and Functional Programming*, 1984.
18. R Glück, J Jørgensen. Generating Optimizing Specializers. *IEEE Computer Society International Conference on Computer Languages*, 1994.
19. R Glück, J Jørgensen. Efficient Multi-Level Generating Extensions for Program Specialization. *Programming Language Implementation and Logic Programming*, 1995.
20. L G C Hamey, J A Webb, I-Chien Wu. An Architecture Independent Programming Language for Low-Level Vision. *Computer Vision, Graphics, and Image Processing*48?:.
21. Fritz Henglein. Efficient Type Inference for Higher-Order Binding-Time Analysis. *International Conference on Functional Programming Languages and Computer Architecture*, 1991.
22. John L Hennessy, David A Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann 1990.
23. Susan Horwitz, Thomas Reps. The Use of Program Dependence Graphs in Software Engineering. *ICSE*, 1992.
24. Adobe Systems, Inc. *PostScript Language Reference Manual*. Addison-Wesley 1990.
25. IRIS Explorer. Numerical Algorithms Group, Ltd 1995.
26. Ulric Jørring, William Scherlis. Compilers and Staging Transformations. *Principles of Programming Languages*, 1986.
27. Stephen C Johnson. YACC - Yet Another Compiler-Compiler. Bell Labs 1975.
28. N Jones, C K Gomard, P Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall 1993.
29. Simon L Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall 1987.
30. Richard Kelsey, Jonathan Rees. A Tractable Scheme Implementation. *Lisp and Symbolic Computation*?:?.
31. Richard Kelsey. Pre-Scheme: A Scheme Dialect for Systems Programming. ?.
32. D Keppel, S J Eggers, R R Henry. A Case for Runtime Code Generation. UW-CSE-91-11-04.
33. D Keppel, S J Eggers, R R Henry. Evaluating Runtime-Compiled Value-Specific Optimizations. UW-CSE-91-11-04.
34. Gregor Kiczales. Towards a New Model of Abstraction in the Engineering of Software. *IMSA*, 1992.
35. Mark Leone, Peter Lee. Lightweight Run-Time Code Generation. *Partial Evaluation and Semantics-Based Program Manipulation*, 1994.
36. Henry Massalin. *Efficient Implementation of Fundamental Operating System Services*. Columbia 1992.
37. Torben Mogensen. *Binding Time Aspects of Partial Evaluation*. DIKU 1989.
38. John Moussouris, Craig Hansen. Architecture of a Broadband Media Processor. *Microprocessor Forum*?:?.
39. Netscape Navigator. Netscape Communications Corporation 1995.
40. John Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley 1994.
41. Frank Pfenning. Logic Programming in the LF Logical Framework. *Logical Frameworks*, 1991.

42. PhotoShop 3.0. Adobe Systems, Inc 1995.
43. Rob Pike, Bart Locanthi, John Reiser. Hardware/Software Trade-offs for Bitmap Graphics on the Blit. *Software-Practice and Experience*15:131-151.
44. QuickDraw GX. Apple Computer, Inc 1995.
45. ScriptX. Kaleida Labs, Inc 1995.
46. Jay M Sipelstein, Guy E Blelloch. Collection-Oriented Languages. *Proceedings of the IEEE*?:?.
47. Richard L Sites. Alpha AXP architecture. *CACM*36:?.
48. Colusa Software. Omniware: A Universal Substrate for Mobile Code. *WWW*, 1995.
49. R M Stallman. *Using and Porting GNU CC*. Free Software Foundation 1989.
50. Richard Stallman. *GNU Emacs Manual*. Free Software Foundation 1987.
51. Guy Steele. *Common Lisp the Language*. Digital Press 1990.
52. Steve Upstill. *The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics*. Addison-Wesley 1989.
53. J E Veenstra, R J Fowler. MINT: a front end for efficient simulation of shared-memory multiprocessors. *Modeling and Simulation of Computers and Tlecommunications Systems*, 1994.
54. Visual Basic v3.0 for Windows. Microsoft 1995.
55. Michael Wolf, Monica Lam. A Data Locality Optimizing Algorithm. *Conference on Programming Language Design and Implementation*, 1991.

This article was processed using the LaTeX macro package with LLNCS style