1 Top

Contents

1.1 Map

This document is organized as follows: after the summary, the first section demonstrates the connection between simple procedures and interpreted languages. The subsequent sections discuss aspects of system design in light of this. Section 8 presents the design of the prototype, and how it can be extended to satisfy the goals. In particular, Section 8.1 defines the intermediate representation and explains its graphical notatioin. Section 9.1 describes on what basis the implementation and examples (including a loop language and 2D graphics) are to be analysed. I conclude with a discussion of the future.

This proposal contains more material than most (including many examples and technical details), but hopefully the hierarchical organization makes it easy to skip the uninteresting material. For example, a reader who already understands the connection between interpreters, procedures, and RTCG may skip subsections 3.1 to 3.3. On the other hand, sections 8.3 to 8.7 present technical details probably only of interest to a partial evaluation specialist.

Please, do not be put-off by the page count.

1.2 Preface

This document is my thesis proposal. My advisor is Peter Lee; the committee also consists of Olivier Danvy, William Scherlis, and Andrew Witkin.

This document was preduced with jar's markup system. It reads its own format and produces HTML and Latex versions. I have concentrated mostly on the HTML end; basically I am writing and organizing assuming a hypertext browser. I apologize for the awkwardness of the paper version.

The HTML version is available at `http://hopeless.mess.cs.cmu.edu:8001/nitrous/top.html`. Postscript from Latex at `http://www.cs.cmu.edu/ spot/proposal.ps`.

1.3 Cute Quote

> Making something variable is easy.
>
> Controlling duration of constancy is the trick.
>
> Alan Perlis, epigram #66

2 Summary

A common strategy used by compilers to optimize programs involves factoring computations out of repeatedly executed procedures, statements, and expressions. For example, some compilers (and some programmers) will ‘hoist’ so-called ‘loop-invariant computations’ out of loops [Dragon]. This basic idea also exists in the area of program transformation, for example in the use of ‘staging transformations’ [JoSche86] that move computations to earlier stages in the execution of the program whenever possible.

Run time code generation (RTCG) is another technique for factoring invariant computations out of repeatedly evaluated expressions. In principle, RTCG can lead to better results than purely static approaches because the ‘invariants’ do not need to be established until run time, when we might expect more useful aspects of the computation to become invariant.

My thesis is that an approach to RTCG exists which will allow one to build software using at a high level of abstraction (by composing many ‘little languages’) while at the same time achieving high performance. I plan to demonstrate this thesis by implementing a prototype system and using it to build several novel graphics applications.

The approach I take to this research involves factoring computations into procedures called ‘generating extensions’ [JoGoSe93]. The classic example of a generating extension is a compiler. For example, consider an interpreter `I`. This is essentially a procedure that takes a program to interpret and that program’s input; the result is the program’s output.

```
I(p, i) → o
```

Typically, the interpreter will be repeatedly applied with the same program but varying inputs:

```
I(p, i1) → o1   I(p, i2) → o2   I(p, i3) → o3
```

and so it often becomes profitable to factor out the computations that are specific to the interpretation of the particular program `p` and perform them only once, leaving behind a program that performs only that part of the computation that depends on the program’s input. The program that performs the factoring and executes the ‘static semantics’ is called a compiler:

```
comp(p) → aout
```

```
aout(i1) → o1   aout(i2) → o2   aout(i3) → o3
```

`Aout` implements the ‘factored out’ computations of `I` applied to `p`.

However, to quote [Abelson92], “The interpreter for a computer language is just another program.” Conversely, any program can be viewed as implementing an interpreter for a language (often a very simple language!), and so this idea of factoring out computations can be applied in all sorts of situations, for example when

```
f(s, d1); f(s, d2); f(s, d3);
```

is replaced with

```
f_s = f_gen(s)
f_s(d1); f_s(d2); f_s(d3);
```

Functional programmers may think of `f_gen` as a ‘smart’ currying function that performs as much of the computation of `f` as is possible when the `S` argument is supplied, then returns a ‘partially applied’ function. If `f` has type `S * D -> T` then `f_gen` would have type `S -> (D -> T)`.

With today’s popular programming systems [CodeWarrior][GCC] many practical differences arise between compilation and RTCG. In the most obvious one, the argument `s` will not be known until run time, and hence the specialized procedure `f_s` cannot be computed until run time. Hence, run time code generation, or RTCG. This also means that the procedures to be factored will often be much simpler than typical interpreters, and so we might hope that the generating extensions might be simpler to implement and also less expensive to execute than the typical compiler.

Still, compilers such as `f_gen` are in practice much harder to port, write, debug, and change than interpreters. And since the cost of compilation has not been a primary concern, it has typically been very expensive to execute a compiler. The cost of compilation is particularly important in situations in which the specialized procedures such as `f_s` might be used only a few times. Reducing the cost of the compilers only exacerbates their implementation difficulties. Thus there have been limited opportunities for RTCG in production code.

Note that several procedures might be composed together (e.g., to implement 'layered' system architectures), and so several stages of 'compilation' might be required in general.

The above motivates a system with the following properties:

fast compilers the generating extensions must run fast.

(semi)automatic make it easy to build generating extensions (compilers) from procedures (interpreters).

composable handle multistage and multilayer systems.

To address these problems, I take an approach based on ideas and techniques developed for *self-applicable partial evaluation* (PE) [JoGoSe93] and apply them to a compiler intermediate representation. PE is a transformation technique that traditionally has been applied to the generation of compilers from interpreters. In other words, PE is an approach for automatically deriving efficient generating extensions. Partial evaluation of mostly-pure Scheme has made great advances in practicality recently, and now it is possible to convert even a higher-order interpreter into a good compiler (whose target language is Scheme) [Jorgensen92].

There's nothing magic about this kind of cogen (compiler generator). It doesn't write any new code, it merely reorganizes the procedures given to it. However, easing the creation of compilers from interpreters makes languages *lightweight*. Such a cogen promises to (and in fact was specifically designed to) alleviate the implementation difficulties of interactive graphics. Thus my title: *Lightweight Languages for Interactive Graphics*.

My system is called *nitrous*. It uses a simple, untyped compiler intermediate representation (IR) as input to and output from a directly implemented compiler generator. Because the input is low-level, few invariants are apparent in the input code. Hints are required to avoid excess specialization and generate good compilers; these hints sometimes take the form of *lifting operators*.

Since the input and output languages are the same, cogen may make several passes over code. This can be used to support *layered languages*, one form of language composition. However, the lifting operators must be generalized to handle multiple passes. Directly implementing cogen instead of creating it by self-applying a specializer has proven to free the system from some peculiar constraints [BiWe92], and obviates the bootstrapping problems of self-application.

There are a number of questions I hope to answer. For example, what are the analytical properties of the transformed code and its execution? How practical is cogen compared to existing systems such as lisp macros? Is the system powerful enough to handle a real language without depending on brittle analyses?

I plan to evaluate the system by implementing several examples, including vector and 2D graphics toolkits. The vector toolkit uses RTCG to convert scalar procedures into unrolled, pipelined loops. In the graphics toolkit, the representation of bitmaps is decoupled from the operations on them. Eg, I use a compiled little language to describe the memory layout of bitmaps. Perhaps the most interesting example is the lift compiler, wherein `cogen` is used to implement itself.

In conclusion, we can summarize this thesis as follows: run time code generation and compilation are two ends of the same underlying optimization. Traditional compilation is great, but fast compilers open up many new and interesting opportunities. Writing compilers manually is hard; we can automate much of it, giving us lightweight languages. My approach is to apply partial evaluation techniques to a compiler IR, yielding *cogen*, a compiler generator designed for RTCG. The approach is proven by the development of vector and 2D graphics toolkits.

3 RTCG and Compilation

The two terms 'run time code generation' and 'compilation' have different connotations though their meaning is really the same. RTCG is fast, low overhead compilation, sometimes for a simple language. The continuity between these terms is explained below, and further demonstrated in the three subsections.

A prototypical interpreter is so general it can be used to compute *any* function; it is Turing complete. But this isn't a hard requirement, witness regular-expression languages. A prototypical language supports inductively structured programs, but since any structure can be encoded into a string (or even an integer, à la Gödel), this isn't a hard rule either. If we stretch the usage of these words, then every program implements an interpreter for a language [Abelson92].

a power procedure base raised to exponent, programs are integers.

a shading procedure phong shading, checkerboard.

a shading language general purpose surface modeler.

The three above examples illustrates this continuity. The generating extension from the `power` example (`power_gen`) is a compiler for a very simple language: programs are just integer exponents. The shading procedure `shade` is more general than `power`; it computes many related functions things depending on the geometric model [note shade-power]. Gradually, as graphics systems grew, the surface properties included simple expression trees [Cook84]. Finally control flow was added and the shading procedure became a shading language.

Another example comes from user interfaces (UIs). A simple UI allows the user to give a sequence of unparameterized commands. This is like a language without any control flow (a flat event stream). A UI with simple record-and-play macros allows procedure call, but without passing parameters. Finally, sophisticated UIs embed a full programming language [Visual-Basic][elisp][SchemePaint][HyperNeWS].

It's interesting that as a program changes over time it typically moves up a chain of generalizations. Eventually it splits and intermediate languages appear.

Any number of advanced programming systems [SML/NJ][Chez][CMUCL] (we refer to these and their ilk as 'lisp systems') support RTCG via `compile` or an equivalent system procedure. The speed of these compilers (and traditional compilers in general) varies tremendously. Mostly it's a function of optimization, but it's also increased by multiple passes, un/parsing, and even file i/o.

Of course, the total run time will *increase* if we spend more time doing code generation itself than we save by the execution of this new and faster code: RTCG wins if $\delta n + \sigma < \iota n$ where n is the number of times we call `power`, δ is the time for the specialized version, σ is the time to generate the specialized procedure, and ι is the time for the interpreted code to execute (given the same exponent value). As n grows, σ loses importance. But when n is small, compilation only works if σ is also small.

If the `power` procedure is viewed as an interpreter, then `power_gen` is its compiler and σ is the compile time. In this case σ can be very small compared to a traditional compiler, but if you call out to `gcc` or use an existing Lisp compiler, σ may be quite large.

3.1 A Power Procedure

Consider the integral power procedure. Each example in this section is presented in Scheme and in C. For code generation the Scheme example assumes the implementation provides a `compile` procedure, and the C example uses the DCG system [DCG].

```
(define (power exp base)
  (if (= 0 exp) 1
    (let* ((n (power (>> exp) base))
           (nn (* n n)))
      (if (odd? exp)
          (* nn base)
        nn))))
```

```
double pow(double base, int exp) {
   int t, bit = 1;
   int square = base;
   double result = 1.0;
   while (bit <= exp) {
      if (bit & exp)
         result *= square;
      bit = bit << 1;
      square *= square;
   }
   return result;
}
```

This code looks clean and efficient, but often it is far from optimal. If the exponent were known to be, eg 20 ($10100_2 = 20_{10}$), you could instead use this specialized version:

```
(define (power-20 base)
  (let* ((base-2 (* base base))
         (base-4 (* base-2 base-2))
         (base-8 (* base-4 base-4))
         (base-16 (* base-8 base-8)))
    (* base-4 base-16)))
```

```
double pow_20(double base) {
   double b2, b4, b8, b16;
   b2 = base * base;
   b4 = b2 * b2;
   b8 = b4 * b4;
   b16 = b8 * b8;
   return b16 * b4;
}
```

Since the control flow, conditionals, and shifting of the original code depended only on the exponent, they can be eliminated. On typical RISC machines, this will run much faster (on a MIPS R3000 the C version ran 2.7 times faster; on an Alpha 2.4 times).

Sometimes the exponent isn't known at compile time, but you do know that, for whichever values arrives later, it will be called many times (perhaps 10,000). This situation is the motivation for run time code generation (RTCG), the synthesis of specialized machine code at run time [note run-time]. Using this technique, a *generating extension* is called with just the exponent. It returns a procedure that finishes the computation:

```
(define (power-gen exp)
  (define (loop exp)
    (if (= 0 exp) 1
      `(let* ((n ,(loop (>> exp)))
              (nn (* n n)))
         ,(if (odd? exp)
              '(* nn base)
            'nn))))
  (compile `(lambda (base) ,(loop exp))))
```

The lisp compiler is called to convert the sexpr data into a procedure. [note removing-times-one].

DCG is an efficient, retargetable, C-callable interface for RTCG [DCG]. The 'D' stands for 'Dynamic' (which is actually a better term than 'run time'). Using it, `power_gen` looks something like this:

```
typedef double (*FPtr)(double);

Symbol loop(int exp, Symbol base) {
   Symbol n, nn;
   if (1 == exp)
      return scnstd(cnstd(1.0));
   n = loop(exp >> 1, base);
   nn = multd(n, n);
   if (exp & 1)
     nn = multd(nn, base);
   return nn;
}

FPtr power_gen(int exp) {
   int ncalls = 0;
   char name[20];
   Symbol base, res;
   base = sargi();
   dcg_param_alloc(&base, ncalls);
   regtree(retd(loop(exp, base)));
   sprintf(name, "power_%d", exp);
   return dcg_gen(sfunc(name), &base, ncalls);
}
```

This compiler will run very fast.

The following sections follow a generalization of this procedure typically occuring in 3D graphics. But this only represents one possible path: in a mathematical library, you might want to provide a more general purpose power procedure that handled negative and then rational exponents. You might also support complex numbers. Such features require additional tests, but as long as they depend solely on the exponent, a compiler can eliminate them.

3.2 A Shading Function

Shading is the computation used in 3D graphics to determine how a surface appears to one's eye, given surface properties and an incident lighting environment. Here is a sketch of a simple shader. It handles specular and diffuse shading, texture mapping, and a checker-board effect.

`Shade` computes how one point on a surface looks to an eye, given a lighting environment. `Surface` is a structure containing the surface properties, typically color, transparency, shininess, and a plastic/metal bit. S, t, and normal give the position and orientation of the surface point. Shade sums the contributions from each of the lights. Each light reflects proportionally to a power of the dot product of the eye and the reflected light.

```
(define (pass-checks? freq s t)
  (odd? (xor (mod (* s freq) 1))
             (mod (* t freq) 1)))

(define (shade eye lights surface s t normal)
  (define (shade-one-light light)
    (let ((k (dot-product eye (reflect (direction light) normal)))
          (exp (roughness->exponent (roughness surface)))
          (diffuse-color
           (cond ((texture? surface) ...)
                 ((and (checked? surface)
                       (pass-checks? (check-frequency surface)
                                     s t))
                  (check-color surface))
                 (else (diffuse-color surface)))))
      (gen-* (color light)
             (gen-+ (gen-* (power exp k)
                           (specular-color surface))
                    (gen-* k diffuse-color)))))
  (gen-+ (ambient surface) ; maybe inside lights?
         (reduce gen-+ (map shade-one-light lights))))
```

Though this code is concise and very abstract, it is cluttered with vector arithmetic and structure references. The vector and color arithmetic is done with generic procedures gen-* and gen-+ that dynamically test the types of their arguments.

When rendering animation, eye, lights, and surface will be fixed for at least each complete frame. If we hold these arguments static and use RTCG, then a loop over the points on a surface can use this program instead:

```
(define (shade-specialized s t normal)
  (let* ((t0 (reflect constant-direction0 normal))
         (t1 (reflect constant-direction1 normal))
         (k (+ (* eye_0 t0_0) (* eye_1 t0_1) (* eye_2 t0_2)))
         (k2 (* k k)) (k4 (* k2 k2))
         (k8 (* k4 k4)) (k16 (* k8 k8))
         (k20 (* k16 k4))
         (m (+ (* eye_0 t1_0) (* eye_1 t1_1) (* eye_2 t1_2)))
         (m2 (* m m)) (m4 (* m2 m2))
         (m8 (* m4 m4)) (m16 (* m8 m8))
         (m20 (* m16 m4)))
```

```
(color (+ a_0
            (* Cl0_0 (+ (* k20 Cs_0) (* k d_0)))
            (* Cl1_0 (+ (* m20 Cs_0) (* m d_0))))
         (+ a_1
            (* Cl0_1 (+ (* k20 Cs_1) (* k d_1)))
            (* Cl1_1 (+ (* m20 Cs_1) (* m d_1))))
         (+ a_2
            (* Cl0_2 (+ (* k20 Cs_2) (* k d_2)))
            (* Cl1_2 (+ (* m20 Cs_2) (* m d_2)))))))
```

All tests, loops, and indirect calls have been eliminated; the remaining code can be scheduled more easily and will run faster. Thus we can more than win back the time spent compiling it in a wide range of pixel resolutions, scene complexities, and numbers of frames.

But a lisp-generating extension for this shader would be much harder to write than the interpreter given above. If this were C/DCG instead of lisp, then the generating extension would be quite tricky for the programmer.

3.3 A Shading Language

The above shader contains only a simple model and this it is not very flexible. It can be generalized by adding more parameters, by making the parameters into tuples, by making them recursive structures, and so on. The generalization process eventually leads to a shader that includes a small interpreter that implements a *shading language*. The core of Pixar's RenderMan(tm) system is exactly this [RenderMan]. Here is a program written in 'the' shading language—a shader—that produces a checkerboard similar to the previous example.

```
surface checker(float Kd         = 0.5,
                      Ka         = 0.1,
                      frequency  = 10;
                color blackcolor = color(0,0,0))
{
   float smod = mod(s * frequency, 1),
         tmod = mod(t * frequency, 1);
   if (smod < 0.5) {
      if (tmod < 0.5)
         Ci = Cs;
      else
         Ci = blackcolor;
   } else {
      if (tmod < 0.5)
         Ci = blackcolor;
      else
         Ci = Cs;
   }
   Oi = Os;
   Ci = Oi * Ci * (Ka * ambient() +
                   Kd * diffuse(faceforward(normalize(N), I)));
}
```

This is *not* C code. This example taken from [RenderMan] page 345.

RenderMan has a number of features that make it particularly easy to write shaders. Short vectors are highly integrated (arithmetic operators work on vectors, and scalars and vectors); it has special global variables (eg `s` and `t` above); it knows how to sum over lights (eg `ambient`, `illuminance`); and can even do derivatives (implemented numerically) with respect to the special variables.

Note that the above program is a (static) *argument* to the shading interpreter, while the code in the previous shading procedure example *is* the interpreter.

Pixar's implementation (`prman`) uses a buffered interpreter (see buffering) to evaluate a shading procedure (which describes how to shade one point on a surface) over a mesh of micropolygons. Pixar's implementation performed very well when it was originally designed (usually within 90% [Peachey94] of compilation, though that was many years ago).

Pixar could have built an implementation that compiled shaders into C, then compiled them with the local C compiler, and finally dynamically loaded the code. But this requires painfully tricky and non-portable code. Furthermore, if the system were interactive the time spent compiling code might introduce an unacceptably long pause.

Recent research from Microsoft indicates compiling shaders and specializing them to particular surface properties has great benefits (as much as 100x) for interactive modeling.

What if Alvy Ray Smith and Darwyn Peachey had had tools like DCG and 'C [tick-C]? The portability (OS and architecture) code could be isolated, while bringing the overhead (σ) way down.

This concludes our discussion of compilation and RTCG. Next, compiler generation via partial evaluation is presented as a tool for achieving lightweight languages.

4 Compiler Generation

The previous section suggests broader use of the fundamental technique of compilation by making fast code generation facilities available at runtime, that is RTCG. But compilers are too hard to write, too hard to change, and not portable enough. Compiler generation attempts to address these problems by automating the process of creating a compiler (as much as is practical) from an interpreter.

How hard is it to write a compiler? It depends on several things. As the source language becomes higher level and the target language becomes lower level, or it has global consistency requirements (such as static types or modules), complexity increases quickly. It is relatively easy to write a compiler that generates lisp, harder to write one that generates C, and harder yet to generate machine code directly. The difficulties are compounded by generating *fast* code.

In addition, it is much harder to change a languages encoded as a compiler instead of as an interpreter. Thus if software is still under development, its interfaces have not settled, or new interfaces are being created, it is easier to work with an interpreter. Thus compiler generation is especially useful for prototyping, exploratory programming, evolutionary programming, and bottom-up programming.

For example, consider the constructioin of a toolkit for artificial life (alife) researchers working with cellular automata (CA). Creating a CA rule corresponds to writing a program. Creating a new class of rules corresponds to creating a new language (eg a Margolus Neighborhood language [CAM]). The researcher is exploring a language space. Interpreters are too slow for interactive, visual-bandwidth calculation. Current practice is to compile CA rules to lookup tables, but this fails for any non-trivial state size, as required by eg reaction-diffusion systems [WiKa91]. User-directed evolution [Sims91] of CAs represents an ideal application for *nitrous*.

What I really want is light(er)weight languages.

Compiler generation attempts to address these problems by automating the process of creating a compiler from an interpreter [someone]. Returning to the `power` example, this amounts to automatically generating `power-gen` from `power` [note coding]:

```
(define power-gen (cogen power '(static dynamic)))
(define power-20 (power-gen 20))
```

Off-line (aka self-applicable) partial evaluation (PE) has matured and now with plenty of foresight and a certain amount of tweeking you can convert even a higher-order interpreter into a compiler (whose target language is Scheme) [Jorgensen92].

This makes languages lightweight. Since languages are generated from their definitions they are as easily modified as interpreters (almost). Furthermore, the interpreter can be used to debug as needed. The complexity of the compiler that originally largely in bookkeeping has been replaced by programmer understanding of binding times, lifting, and termination issues (picking bookkeeping strategies).

5 Alternatives

This section compares my approach to some existing alternatives techniques. This could be considered either 'pre-design' work, or 'related work'. Precompilation and buffering are two other ways of working around the performance problems of interpreters. Instead of implementing cogen directly, it's possible to generate it using a self-applicable partial evaluator. Finally, lisp systems with macros are considered.

1. precompilation vs RTCG
2. buffering and APL
3. self application and cogen
4. lisp and macros

5.1 Precompilation vs RTCG

In the power example, instead of generating `power-20` at run time you could compile many different power procedures, and dispatch at run time. This is *precompilation*: a space-time trade-off where a lookup table with programs for table entries replaces a more general program.

But which values do you precompile for? For `power`, you could probably use the first 100 integers pretty well, but in general, success depends on predicting the distribution of the dynamic values. As the language becomes more complex, its parameter space grows, and its use becomes more dynamic, the code space required to precompile it explodes exponentially. Pike's bitblt handled 1944 ($2^3 3^5$) different cases [blit]. The same effect is visible in most any math vector library [CVL][NAG], or in my Shape language [Shape].

The precompiled procedures can be hand-written for speed or automatically generated. Hand-writing requires duplicate code, but allows for tweeking. Automation requires the same compiler as RTCG requires.

Though all too often the meta-language used is `cpp` or `awk` [links!], in any language with a decent macro-system (metalanguage), and lisp in particular, precompilation is in fact a fundamentally important technique [On-Lisp].

5.2 Buffering and APL

APL is a programming language for numerical matrix work [APL]; it is the original *collection oriented language* [SiBle91]. This section examines the use of *buffering* (a term from operating systems) to implement such languages on uniprocessors with a memory hierarchy. Buffering is a very effective and widely used programming technique, but also has its limits: latency suffers, sometimes performance suffers, dynamic conditionals induce copying and increase overhead. There is no known way to effectively handle dynamic jumps, or higher order scans, reduces, and permutations.

If a program is applied to many data:

```
vector_eval(struct program, vector v) {
  for (i = 0; i < max(v); i++)
     scalar_eval(program, v, i);
}

scalar_eval(struct program, scalar v) {
  switch(program) {
    case p_prim:
     prim(v);
  }
}
```

then the control flow for the interpreter and the loop over the data may be interchanged. This is buffered:

```
vector_eval(struct program, vector v) {
  switch(program) {
    case p_prim:
     for (i = 0; i < max(v); i++)
        prim(v, i);
    ...
  }
}
```

Rather than loop over *all* the data, it can be advantageous to limit the loop length (block size) so that the total temporary memory required fits within the (or a) cache. Renderman and Nyquist [Nyquist] do this. This is called blocking or *tiling* [WoLa91]. Note that as the temporary space required increases, the vector length must go down to remain in the cache, though this probably only makes a difference in extreme cases.

However there are some drawbacks to buffering. Latency suffers because a complete block must be processed before the first result is available. Latency is particularly important in interactive music applications. Throughput may suffer from increased cache and memory traffic, or from an inability to effectively use multiple functional units.

Dynamic control constructs are very difficult to handle while buffering. Conditionals can be handled either with mask vectors or by copying and collapsing. I don't know any reasonable way to handle dynamic jumps, or higher-order scans, reduces, and permutations.

With RTCG the static control flow of the inner interpreter is eliminated. The resulting loop may be better optimized. Memory references for temporaries in the program can be replaced with register references. Tiling is still useful, though not as often.

See [ThoDa95] for an analysis of buffering vs compilation in the context of audio signal synthesis.

5.3 Self Application vs Cogen

The nitrous cogen design is descended from the Mix system [????], via Similix and Schism. These three systems all consist of self-applicable partial evaluators for various languages. That is instead of providing `cogen` they provide `specialize` where `((specialize f s) d) = (f s d)` and then, according to the 3rd Futamura projection [Fu78]:

```
(define cogen (specialize (specialize specialize)))
```

Such systems are also known as *off-line specializers*. An *on-line specializer* satisfies the definition above, but doesn't produce an efficient cogen. [FUSE] is the canonical on-line system.

More recent systems such as [Fabius] and [BiWe93] implement cogen directly instead of generating it. This was originally motivated by the type-encoding problem of an ML specializer, but is otherwise useful. For example, cogen doesn't have to be written (or compiled into) its own input language (though in theory, ultimately this is only a bootstrapping problem).

The disadvantage of a direct implementation of cogen is that metastatic and static values have seperate but parallel implementations, thus a specializer may ultimately be simpler.

5.4 Lisp and Macros

in lisp, macros are like special sub-languages that compile into lisp.

Backquote compilation is easily handled by cogen. It appears that at least the matching and rebuilding features of `syntax-rules` (see appendix to [r4rs]) can be implemented efficiently with cogen. How does cogen's variable renaming relate to hygenic macros?

There are several typical ways of extanding lisp: allowing the user to defining new procedures, allowing the user to define new syntax with macros, allowing user to define new lexers with read macros. cogen is like allowing the user to define new 'evals'.

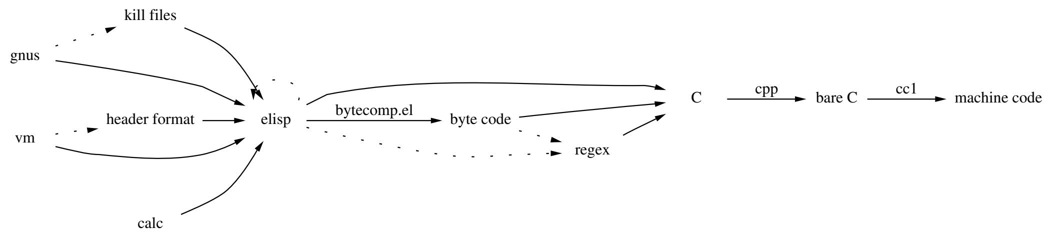
CLOS uses 'mutable' reified structures, allowing you to customize the language [MOP]. cogen allows the creation of new languages, with exactly the properties you need. See also [open-comp].

6 Composing Languages

How does compiler generation work when a system is composed of many languages? Organizing and specifying systems with interfaces is a well-known software engineering technique. But interfaces, languages, and procedures are all really the same thing [Lampsons-hints]. In this section we examine how multiple languages are assembled into systems, and how this impacts compiler generation.

6.1 Emacs

Emacs is a good example of a system with many languages: C code is run through cpp then compiled into machine code. Elisp is compiled into byte-code then interpreted. Regular expression matching is available as an elisp primitive. Modes implement UI languages, and some modes even have full-scale interpreters built in [note emacs-languages-all]. These relationships form a directed graph [this graph isn't quite right. should delete killfiles and header format, dotted edges, but what is elisp's eval?].



6.2 Reflection

Say interpreter `int1` has another interpreter `int2` as a primop. `int2` is an embedded language. In emacs, this is the relationship of bytecode to the regexp language. Schematically, the code looks like this:

```
int2(prog, data) =
  ...
```

```
int1(prog, data) =
  switch prog
    case: int2(data1, data2)
    ...
```

Now say that for some program `p` calls `int2` with the same `data1` again and again, ie there are three stages: `prog`, `data1`, and `data2`. `Data1` can be compiled by adding another case to `int1`:

```
int2(prog, data) =
  ...

comp_int2 = cogen(int2, (s d))

int1(prog, data) =
  switch prog
    case: comp_int2(data1)
    case: apply(x, data2)
    ...
```

Here `comp_int1 = cogen(int1, (s d))` works fine.

What happens if `int1` and `int2` are the same? This is *reflection*. A lisp system's `eval` is a familiar example. A fixed point is required to generate the compiler: it is closed because cogen memoizes on binding times (the table is stored in `eval`) (note: it has to look it up in the table every time it is called, here we see another artifact of direct cogen instead of self-applicataion).

In general, reflective sublanguage relationships form a directed graph. This graph is lazily traversed by `cogen`.

6.3 Layers

Consider a different kind of composition:

```
int1(prog1, data1) =
  ...

int2_1 = '(some program text)

int2(prog2, data2) =
   int1(int2_1, (list prog2 data2))
```

That is, `int2_1` is a program written in the language *defined by* `int1`. We say `int2` is a *layer* on top of `int1`. Here `cogen(int2 (s d))` fails because `int2_1` is represented with data instead of code, so it doesn't get very far as a metastatic value. So instead write:

```
comp_int1 = cogen(int1, (s d))

obj = comp_int1(int2_1)

int2(prog2 data2) =
   obj(prog2, data2)
```

Now in `cogen(int2, (s d))` `obj` is a procedure so it is analyzed by cogen properly. Since various annotations are required for most interesting inputs to cogen, `obj` must in general contain annotations. These annotations must be created by cogen from `int1` and `int2_1`.

The above is equivalent to using a binding time lattice with multiple stages.

7 Related Work

A number of other research initiatives are developing these same ideas:

DCG compilers hand-written in C generate portable IR; DCG translates the IR to real code. Requires about 350 instructions to generate one instruction. Backends for different architectures are actually generated using a special-purpose language. [DCG].

'C a extension of C built with DCG, provides high level interface [tick-C]. Basically like Fabius for C with hand-annotation.

Similix off-line specializer for simplified Scheme. Compilers generate Scheme. Not multistage, polyvariant specializer but monovariant BTA; only supports simple lifting, but runs fast [Similix][Similix-doc].

Schism similar to Similix, but handles types and polyvariance. Supports *filters* for more control of lifting among other things. Source programs can be either ML or Scheme [Schism].

Fabius is a compiler generator for a simplified first-order ML. The compilers produce machine code directly, thus they are very, very fast [Fabius].

Washington a cadre at the University of Washington have been studying the performance of manual RTCG. They compare a template compiler that uses tens of instructions to generate each instruction and a simulated IR system to traditional compilation [KeEggeHe91][KeEggHe93].

Fuse is a graph-based on-line PE for Scheme. It can compile code, but not generate compilers. Designed primarily for scientific computation [Fuse]

Synthetix not clear what it will be, but a multi-stage compiler generation is also one of their goals. They are concentrating on operating systems, and file systems in particular [Synthetix].

qua-C a on-line specializer for Alpha machine code [YaSa]. Uses hints. A number of complications are introduced by analyzing real code, though this allows code from any language/compiler to be processed. Still work in progress.

None of the automatic systems accept a low-level IR language.
None of these systems supports multilayer evaluation.

The IR used by DCG is described in [FraHa91]. They make a big deal of the difference between their procedure call interface passing DAGs, and so-called 'Abstract Machines' (AMs). They confuse the nature of the data structure representing the code (it's trivial to convert my abstract machine into DAGs by replacing the environment with pointers) with using higher-order call interface instead of a one-way stream of bytes.

So how do these systems reduce σ? DCG accepts a compiler intermediate representation consisting of DAGs of primops [DCG] and C function calls. Using a lower-level language reduces the time required to assemble it into finally executable machine code. Fabius generates code directly, for even lighterweight compilers [Fabius].

Systems such as [Self][Smalltalk] have used lightweight code generation in the form of feedback or lazy compilation.

8 System Design

My system is called *nitrous* [note name]. This section describes its design and how the goals influence it. The design is based on the techniques developed by the off-line (aka self-applicable) Partial Evaluation (PE) community [JoGoSe93]. The following subsections provide detailed technical information.

1. root language
2. interface to `cogen`
3. binding time analysis
4. generation pass
5. variable splitting
6. lift compiler

Currently Nitrous is implemented with scheme48 [scheme48], an efficient bytecoded Scheme. Scheme is very good for prototyping languages.

The rest of this section serves as a summary of and map to the subsections.

The easiest way it to make cogen composable is to make make cogen accept the same language as it produces. I call this language `root`. To facilitate fast compilers `root` is just an abstract machine language (see root for more).

The compiler generator itself is called `cogen`. It takes a procedure and a binding time pattern as arguments. The binding time pattern specifies which of the interpreter's arguments are the program (`S`) and which are the data (`D`) (see interface for more).

How does `cogen` work? It makes two passes over the interpreter. The first pass factors the program into the generating extension and the residual code, this pass is called *binding time analysis* (see bta for more). There are two main ways of implementing BTA: with type inference [Henglein91] or via abstract interpretation [Consel93]. The prototype currently is oriented towards the latter, though an inference system may eventually prove better.

The second pass (usually called 'specialization', though 'generation' is more accurate here since this is a direct cogen) recursively traverses the `code` structures of the interpreter and converts each into a compiler (generating extension). *Static* instructions are copied into the compiler as they are. *Dynamic* instructions appear as templates: they produce the output from the compiler (see gen for more).

There are three environments (envs) to keep straight: the metastatic env, the static env, and the dynamic env. The metastatic env (also called the BT env, or the analysis env, or cogen's env) exists in the abstract interpretation, it maps all the program's variables to binding times. The static env (also called the compiler env) exists in the compiler, it maps the static variables to values. The dynamic env (also called the residual env) exists in the generated code. Note that if cogen is applied to an interpreter with an environment, then there are four.

Both cogen itself and the compilers it produces are memoizing. Memoization is used to avoid infinite programs by producing loops. Cogen memoizes on binding times. Generated compilers memoize on static values. (see envs) (see gen) (see layers).

Variable splitting is a technique for eliminating unnecessary intermediate data structures in programs. While this is a generally useful optimization, it is a necessity in PE if environments are to be treated efficiently (see splitting for more).

Inlining is another generally useful optimization that is required to get decent results from PE.

It is important that variable splitting and inlining *not* just be handled as general post-pass optimizations; that would be too slow. Instead specialized versions of these optimizations will be integrated into the generated compilers.

Side-effects are handled simply by forcing them all to runtime, thus compilers must be purely functional. This is an important deficiency. It appears likely that analysis can overcome this [Heintze94].

In summary, the design objectives and how they are satisfied are:

fast compilers compilers produce (and the backend accepts) abstract machine code.

semiautomatic directly implemented cogen based on partial evaluation converts interpreters into compilers with minimal human direction.

composable since cogen's output language is the same as its input language both multiple layers of interpreters and multiple stages of compilation are handled.

powerful nitrous already handles higher-order jumps, polymorphism, data structures, and dynamic side-effects. At least modules, exceptions, and compile-time side-effects remain.

predictable since I have no fear of annotation and binding time assertions, the results and termination of cogen should be understood by the programmer.

8.1 Root Language

The root language (input to and output from cogen) is a untyped CPS-CPS (continuation passing style, closure passing style) language [Appel]. You could also describe it as an unlimited-register abstract machine code. It should be easy to assemble it into executable code, allowing fast generated compilers.

Here is a quick definition of the root language's syntax and semantics:

```
code ⟶ (code name args instructions)
instruction ⟶ (prim v prim . args)
              | (const v c)
              | (exit v)
              | (if v true-branch)
              | (jump v . args)
v ⟶ variable
true-branch ⟶ instructions
instructions ⟶ instruction list
args ⟶ variable list
c ⟶ constant
name ⟶ constant
prim ⟶ lambda-value
```

```
(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))))))))))
```

[note match]

Note that in fact only `exit` and `jump` may appear in final position in instruction lists.

A `code` structure is like a procedure because it has a formal parameter list, but instead of returning a value, it simply jumps to another `code` structure. Allocation for activation records is explicit.

I chose an untyped language because types complicate partial evaluation [BiWe93].

I chose a continuation passing language over a procedure-call language for several reasons:

- to propagate static information past conditionals without techniques described in [writing-mix-in-cps]
- may handle other control structures, or at least provide a framework for handling them
- superset of many high-level languages.

Here is some example programs in the root language: 8.1.1 and 8.1.2.

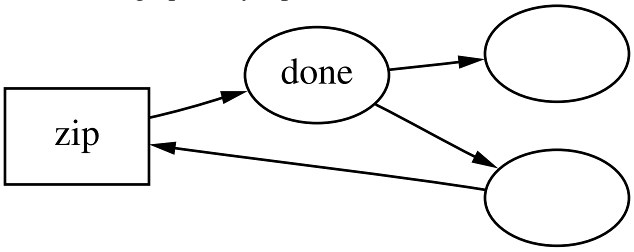
8.1.1 Eg Zip

This root program defines a procedure that builds an environment from lists of names and values.

```
(code zip
      (vars vals r k)
      ((prim done null? vars)
       (if done ((prim p car k)
                 (jump p k r)))
       (prim vars-hd car vars)
       (prim vals-hd car vals)
       (prim vars-tl cdr vars)
       (prim vals-tl cdr vals)
       (prim bind cons vars-hd vals-hd)
       (prim spine cons bind r)
       (const zip (code zip ...))
       (jump zip vars-tl vals-tl spine k))
      ...)
```

Being recursive, `zip` calls itself; the value in the `(const zip (code zip ...))` instruction is a circular pointer. This avoids an explicit recursive environment construct, or a special top-level environment. The final `...` in the stucture hides the memo table used by cogen.

The code is graphically represented like this:



Code structures start with a rectangle labeled with the name of the code. Each straight list of instructions is represented with an oval. Lists ending with an `if` instruction are labeled with the name of the tested variable, those ending with a `jump` are linked with a solid edge to the called code. If the jump target is unknown (as when a procedure returns) then the oval is terminal.

(see envs)

8.1.2 Compose

This section demonstrates how higher-order `root` code works, how `root` is created with Scheme, and more about how it is displayed graphically.

The following Scheme procedure:

```
(define (compose f g) (lambda (x) (f (g x))))
```

is equivalent to the `root` program created by the following Scheme code:

```
(define compose
  (car (code-rec
        `((code compose
                (f g k)
                (,@(make-closure 'apply-g '(f g) 'ag)
                 ,@(call 'k '(ag))))
          (code apply-g
                (ag x k)
                (,@(unmake-closure '(f g) 'ag)
                 ,@(make-closure 'apply-f '(f k) 'af)
                 ,@(call 'g '(x af))))
          (code apply-f
                (af r)
                (,@(unmake-closure '(f k) 'af)
                 ,@(call 'f '(r k))))))))
```

`make-closure`, `unmake-closure`, and `call` are Scheme procedures used to simplify writing higher-order code:

(make-closure *code-name save-locals result-name*)

(unmake-closure *restore-locals closure*)

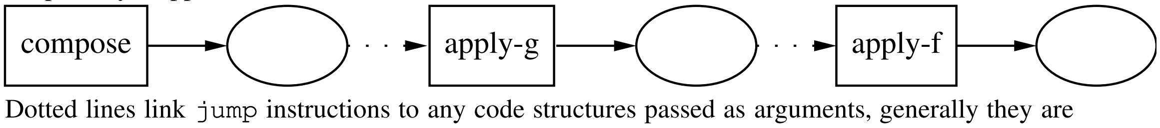
(call *closure args*)

`code-rec` makes the small changes required to convert sexpr syntax into `code` structures. It's most important task is to ease building recursive procedures by replacing *names* of codes with the actual values.

The above Scheme code expands into the following `code` structure. `Compose` creates and returns a closure (`ag`) made of `apply-g` and `f` and `g`. [more?]

```
((code compose
       (f g k)
       ((const cnstr#-16 ())
        (prim cnstr#-15 cons g cnstr#-16)
        (prim cnstr#-13 cons f cnstr#-15)
        (const p#-14 (code apply-g ...))
        (prim ag closure-cons p#-14 cnstr#-13)
        (prim code-pointer#-17 car k)
        (jump code-pointer#-17 k ag))
       ...)

 (code apply-g
       (ag x k)
       ((prim dstr#-18 cdr ag)
        (prim f car dstr#-18)
        (prim dstr#-19 cdr dstr#-18)
        (prim g car dstr#-19)
        (prim dstr#-20 cdr dstr#-19)
        (const cnstr#-24 ())
        (prim cnstr#-23 cons k cnstr#-24)
        (prim cnstr#-21 cons f cnstr#-23)
        (const p#-22 (code apply-f ...))
        (prim af closure-cons p#-22 cnstr#-21)
        (prim code-pointer#-25 car g)
        (jump code-pointer#-25 g x af))
       ...)

 (code apply-f
       (af r)
       ((prim dstr#-26 cdr af)
        (prim f car dstr#-26)
        (prim dstr#-27 cdr dstr#-26)
        (prim k car dstr#-27)
        (prim dstr#-28 cdr dstr#-27)
        (prim code-pointer#-29 car f)
        (jump code-pointer#-29 f r k))
       ...))
```

Graphically it appears:



Dotted lines link `jump` instructions to any code structures passed as arguments, generally they are continuations or closures.

8.2 Interface

The compiler generator itself is called `cogen`. If the interpreter is called like this:

```
(define tlc top-level-continuation)
(define ans (eval-root interp (list program arguments tlc)))
```

Then you could compile `program` and compute the same `ans` like this:

```
(define binding-time-pattern '(static dynamic dynamic))
(define compiler (cogen interp binding-time-pattern))
(define fast-prog (eval-root compiler (list tlc program)))
(define ans (eval-root fast-prog (list arguments tlc)))
```

The binding time pattern specifies which arguments to the interpreter are the program (static) and which are the data (dynamic). Note that generated compilers take their continuation as their first argument (this should probably change). An additional continuation is introduced because the compiler has to return somewhere.

`Cogen`'s `interp` argument is a `code` structure [note coding]. `cogen` slightly extends the semantics of the `root` language to include the semantics of hints, lifting, and a number of primops.

The primops are convered in section 8.3 except `apply`, which is used to create interpreters with an open-ended set of primops.

It also accepts lift directives (syntactically they appear like instructions). Lifting is required to avoid excessive specialization and non-termination. Obviously required lifts are handled automatically, but manual annotation is still often required when writing programs directly in `root`.

[example simple lift directive, mention complex lifting]

8.3 Binding-Time Analysis

The purpose of binding-time analysis is to factor the program into the generating extension and the residual code. The analysis is described here as an abstract interpretation [NN?], with binding times as abstract values. It may be more efficient to implement as a type inference [Henglein91].

The analysis starts with a root language code structure and the binding times of its arguments. The simplest binding times are static (`S`) and dynamic (`D`). Static arguments are available to the generated compiler, and are thus considered 'program'; dynamic arguments appear as arguments to the code returned by the compiler.

Since there are no infinite ascending chains in the binding-time lattice, termination of cogen is guaranteed.

The set of binding times (the lattice) is described in five steps, each elaborates the previous in order to capture more static information.

1. <u>simple</u>
2. <u>pairs</u>
3. <u>first-order jumps and constants</u>
4. <u>higher order jumps</u>
5. <u>spines</u>

The final, formal definition of the binding times:

```
bt ⟶ S | D | (cons bt bt cp is-clo is-sure) | (const v)
cp ⟶ instruction
is-clo ⟶ boolean
is-sure ⟶ boolean
v ⟶ any-value
```

I haven't finished the formal definition of the ordering of the lattice. Here's a first approximation:

```
(const v) ⊏ S ⊏ (cons a d ...) ⊏ D
(cons a1 d1 cp1 ...) ⊑ (cons a2 d2 cp2 ...) ⇔
    a1 ⊑ a2 and d1 ⊑ d2 and cp1 = cp2
```

8.3.1 Simple

The simplest BT lattice has just two elements:

```
bt ⟶ S | D
```

dynamic is top, static is bottom (S ⊏ D).

Say we are analyzing

```
(prim r + a b)
```

In abstract interpretation, the value assigned to r is the abstraction of the + primop (denoted $\tilde{+}$) applied to the values of a and b in the analysis environment.

In my BTA the abstraction of all the generic primops is the same: if any argument is not S then the result is D. Call this function $\widetilde{prim}$. Thus if any argument is unknown at compile time, then the prim must be delayed until runtime.

8.3.2 Pairs

Say the binding time of s were S and d were D in the following code:

```
(prim p cons s d)
(prim x car p)
```

and assume cons were treated as a generic primop (ie $\widetilde{cons} = \widetilde{prim}$) then x would be D, even though we can deduce its value at compile time by using the semantics of cons and car.

To do this, first allow *partially static* binding times:

```
bt ⟶ S | D | (cons bt bt)
```

and define $\widetilde{cons}$ and $\widetilde{car}$ specially:

```
cons~ S S = S
        | D D = D
        | a d = (cons a d)

car~ S = S
     | D = D
     | (cons a d) = a
```

In the above example, p would then be assigned the partially static binding time (cons S D), and x would be S. $\widetilde{cdr}$ is defined similarly.

These new binding times are ordered as follows to make a lattice:

```
S ⊏ (cons a d) ⊏ D
(cons a1 d1) ⊑ (cons a2 d2) ⇔ a1 ⊑ a2 and d1 ⊑ d2
```

`S` and `D` are still bottom and top, but between them lie the partially static binding times.

Ok, that much is easy, but it has a serious problem: the lattice now contains infinite ascending chains. Eg

```
S ⊏ (cons S D) ⊏ (cons (cons S D) D) ⊏
    (cons (cons (cons S D) D) D) ...
```

This happens if the source program recurses over a potentially unbounded data structure. The BTA could detect this conservatively and lift the offending value to `D`, but this loses too much information (see envs). Instead, directed graphs are used to abstract partially static values of unbounded size.

Since there are only a finite number of `cons` instructions in the source program, one of them must be repeated in the BTs of an ascending chain. Such a BT can be *widened* [Cousot] by merging the duplicate nodes, and joining the children. We label the cons nodes so that duplicates can be identified:

```
bt ⟶ S | D | (cons bt bt cp)
cp ⟶ instruction
```

`Cp` stands for *cons point*, it is represented with a pointer to the `root` source instruction where this cons is created. Cons points are denoted with the name of the variable bound to the cons cell, eg the binding time of `p` above is written `(cons S D p)`.

Thus if two nodes with the same cons point (say `(cons a1 d1 p)` and `(cons a2 d2 p)`) appear in a binding time, then that BT can be widened by replacing them with the single node `(cons (⊔ a1 a2) (⊔ d1 d2) p)`.

The formal definition of the ordering of the lattice is now:

```
S ⊏ (cons a d ...) ⊏ D
(cons a1 d1 cp1) ⊑ (cons a2 d2 cp2) ⇔
a1 ⊑ a2 and d1 ⊑ d2 and cp1 = cp2
```

By collapsing all nodes with duplicate cons-points, infinite ascending chains can be avoided, guaranteeing termination (see envs).

One of these new binding times is a graph grammar dividing a data structure into static and dynamic portions. Sometimes an atom is identified as partially static by the grammar, but this is allowed since in abstract interpretation we only need *approximate* actual values, and any partially static value approximates a purely static one.

The quality of an abstraction is measured by how well it matches the distinctions required between typical programs. Grammars have proven tremendously effective in capturing information about data structures with finite descriptions. [Mogensen89] first applied grammars to partially static structures, the techniques was later refined in [Consel90] and [Consel93].

Implementations of partially static structures usually have to be careful to avoid the following unsound reduction:

```
(cdr (cons (printf D) 23)) → 23
```

But in continuation-passing style, arguments to prims are variables not expressions, so there is no danger in eliminating them. Instead I plan to do all dead code elimination in the backend.

8.3.3 First-order Jumps and Constants

A typical first-order jump looks like

```
(...
 (const p (code somewhere ...))
 (jump p a1 a2))
```

Cogen needs to know the destination of the jump in order to generate an extension for it (see jumps). Thus the code-pointer must be in the binding time. The lattice can be extended to include constants (though only constants that are code-pointers are needed for jumps, other constants come in handy):

bt ⟶ `S` | `D` | `(cons` *bt bt cp*`)` | `(const` *v*`)`
cp ⟶ *instruction*
v ⟶ *any-value*

BTs that are constants are called *metastatic* because when a specializer is self-applied they are static for the outer specializer (the one that is analyzing the inner specializer).

Formal ordering requires an additional rule:

`(const` *v*`)` $\sqsubset$ `S`

Since metastatic values are incomparable, this doesn't introduce any infinite ascending chains.

8.3.4 Higher-order Jumps

Consider this code:

```
(...
 (const code-pointer (code ...))
 (prim closure cons code-pointer bound-vars)
 (jump p closure))
```

Say `p` is `D` and `bound-vars` is partially static. Then normally `closure` would be lifted to `D` before the jump. But if we know `closure` will only be called, then we can do better: replace it with a closure where `code-pointer` has been specialized to the static portion of `bound-vars`.

Rather than automatically recognizing this opportunity or using a typed language, nitrous relies on a hint in the code: `closure-cons` marks a particular cons cell as a closure.

bt ⟶ `S` | `D` | `(cons` *bt bt cp is-clo*`)` | `(const` *v*`)`
cp ⟶ *instruction*
is-clo ⟶ *boolean*
v ⟶ *any-value*

Unlike other higher-order partial-evaluators, specialization of the code pointer to its bound variables does not happen at the point in the source where the lambda (`closure-cons`) appears. Instead, the full structure is propagated as a normal value until its binding time is lifted to `D`.

Cogen will create an extension for code structure *cd* against partially static binding time *p* when it encounters the following lift:

`(cons (const` *cd*`)` *p* `_ #t)` → `D`

The compiler calls this extension to preserve the static information in `p` rather than lose it to the lift.

`closure-cons` guarantees to cogen that the cell will only be used in a particular way (jump to the car, passing the cell itself as the first argument), thus it defines a higher-order calling convention. If the programmer used a different representation, cogen would not be able to follow the control flow. This is one way the system is brittle.

This is somewhat different from how lambda expressions are handled in [Consel90].

[note higher-order-cruft]

8.3.5 Spines

The result of `null?`, `pair?`, or `atom?` may be `S` or even metastatic even if its argument is partially dynamic. `cogen` handles them specially to avoid lifting to `D`.

Consider the `length` procedure applied to a typical partially static alist (as in section Environments). Somewhere it makes this test:

```
(code length
      (l n k)
      ((prim t atom? l)
       (if t ((return ...)))
       (loop ...)))
```

`L` is partially static, so if $\widetilde{atom}$? = $\widetilde{prim}$ then `l` would be lifted to `D`, so `t` would be `D`, and we lose. Instead, the generated compiler can test the static value. This requires no modification to the binding time lattice, only implementation of $\widetilde{atom}$?.

Whereas a human programmer would be unlikely to write code having a test with a *metastatic* result, such code could easily be automatically generated by a higher layer or the lift compiler (see lifting). Extending $\widetilde{atom}$? to have metastatic results requires adding another bit to the binding times for conses:

```
bt ⟶ S | D | (cons bt bt cp is-clo is-sure) | (const v)
cp ⟶ instruction
is-clo ⟶ boolean
is-sure ⟶ boolean
v ⟶ any-value
```

If a cons has never been joined with `S`, then it is guaranteed to be a cons, and no longer approximates a static atom:

```
(cons a d cp ic #t) ⊔ S → (cons a d cp ic #f)
```

This is a big change to the lattice, but I think it's still safe. A formal definition of the ordering of the lattice and a soundness proof remain. Its full ramifications for cogen are not completely understood.

You could also just not reduce such things, and leave them as lub (or) nodes in the binding times. I did this originally, and this is basically what [Mogenson89] does: represent binding times with grammars instead of with graphs (only real difference is garbage collection and normal form properties). Simlix and Schism don't have this bit.

8.4 Generation

This section describes how cogen uses the binding time information to produce the compilers. It's surprisingly easy. First we look at what cogen does with a simple procedure. Then we examine partially static instructions, then metastatic values, and finally the how control flow is handled in general.

8.4.1 Simple

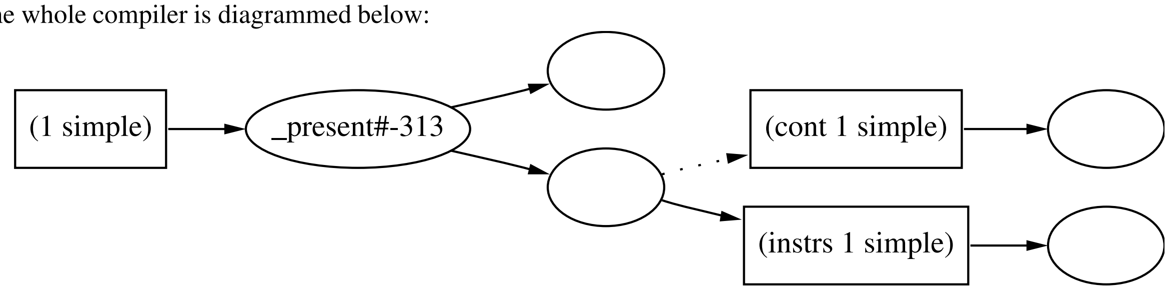
Apply `cogen` to this procedure with binding time pattern of `(S D D)`:

```
(code simple
      (x y k)
      ((const c 10)
       (prim s1 + c x)
       (prim s2 + s1 y)
       (jump k s2))
      ...)
```

The core of the resulting generating extension is a code that executes static instructions, conses up a list of residual instructions, and finally reverses and returns the list.

```
(code (instrs 1 simple)
       (k#-307 x)
       ((const a#-308 ())
        (const c 10)
        (prim s1 + c x)
        (prim i#-309 (lambda (v) `(const s1 ,v)) s1)
        (prim a#-310 cons i#-309 a#-308)
        (const i#-312 (prim s2 + s1 y))
        (prim a#-311 cons i#-312 a#-310)
        (const i#-313 (jump k s2))
        (prim a#-314 cons i#-313 a#-311)
        (prim rev#-315 reverse a#-314)
        (prim p#-317 car k#-307)
        (jump p#-317 k#-307 rev#-315))
       ...)
```

The whole compiler is diagrammed below:



The entry point is named (1 simple). Before it generates any code it checks its memo table to see if it has specialized to these static values before. If so, it just returns its previous result; this is how loops are generated.

Next (instrs 1 simple) constructs the residual instruction list, as above.

Finally (cont 1 simple) wraps the instructions in a code structure, and stores the code in the memo table. If assembly into machine code were supported, it would happen here.

8.4.2 Pairs

Simple instructions are either executed in the compiler, or generated unmodified. As described in this section, if an instruction has static and dynamic parts, it must appear in both the compiler and the generated code.

Here are examples of how some partially static instructions specialize. Given these bindings in the metastatic environment.

```
s → S
d → D
p → (cons S D)
q → (cons (cons S D) (cons S D))
```

If the first column appears in an interpreter, the second column is the static output (code executed by the compiler), the third column is the dynamic output (residual code produced by the compiler). Note that id is the identity primop.

```
source                  compiler             residual
(prim r cons s d)       (prim r id s)        (prim r id d)
(prim r car p)          (prim r id p)
(prim r cdr p)                               (prim r id p)
(prim r cons p d)       (prim r id p)        (prim r cons p d)
(prim r car q)          (prim r car q)       (prim r car q)
```

A cons cell is eliminated at compile time if either branch is purely `D`. A cons cell is eliminated at run time value if either branch is purely `S`. Eg if the car of a cons is pure `D`, then that argument will not exist in the compiler, so the cons is replaced with a simple assignment to the cdr.

Note that we are generating stupid code: this `id` prim is just a renaming. It would be better to avoid it entirely (see splitting).

8.4.3 Jumps

There are two possibilities when cogen reaches a jump instruction: the destination is known or unknown.

If the binding time of the destination is metastatic, then the destination is known. In this case, cogen calls itself recursively to create an extension. The generated compiler calls this extension (passing the static arguments) and returns to a `code` (labeled `done-with`) that finishes building the jump, reverses the instruction list, and returns.

Otherwise the destination is unknown, so all the arguments are lifted to `D`, and a residual jump is generated.

8.4.4 Constants

If a conditional's predicate is metastatic, then the generated compiler only handles one branch.

To simplify the implementation, cogen tracks constants in the binding times, but the same constants are duplicated in the compiler. Since removing the metastatic values from the compiler is the same as removing the static values from the generated code, this is an artifact of the direct implementation of cogen instead of using self application. Thus could a specializer eliminate the code-pointer from closures of 'known' procedures?

[note jump-conversion]

8.4.5 Other

In general, for each code block in the input to the compiler generator, the following output code blocks are created: the entry point (which checks the memo table), the exit code (which memoizes and builds the result), one for each linear list of instructions (which does the static instructions and accumulates a list of dynamic instructions), plus another if it ends in a metastatic jump (to generate a jump to the code just generated), plus two more for each dynamic conditional (to recursively build a tree of instructions).

Lifts are either handled as a simple instruction (trivial lifts), with a recursive call (a higher order lift thus produces an additional code block), or by modifying the source to make a call to a lifter (a procedure returned by the lift compiler (see lifting), probably producing lots more code blocks).

8.5 Variable Splitting

Variable splitting is a technique for eliminating unnecessary intermediate data structures in programs. It is also known as retyping [ref] and arity raising [ref], and it's related to unboxing[ref]. Eg

```
(define plus (lambda (p) (+ (car p) (cdr p))))
(define dumb (lambda (x) (plus (cons (sin x) x))))
```

could be replaced with

```
(define plus1 (lambda (p1 p2) (+ p1 p2)))
(define dumb (lambda (x) (plus1 (sin x) x)))
```

Though this is a general-purpose optimization, like inlining it is critically important to the quality of code produced by PE (see envs).

I plan to implement this in `cogen` by making the following changes:

1. The binding times are unaffected.
2. The static environment and static code in the generated compiler are unaffected.
3. Augment the compiler's static environment with a *shape* environment. Variables with a dynamic component have a value in the shape environment.
4. A value in the shape environment is initialized to the name of the variable that will contain its dynamic value.
5. If any one of the arguments to a dynamic prim is not atomic, then lift it to pure `D` by generating cons instructions according to the shape.
6. The dynamic portion of a cons/car/cdr is now nothing, instead the compiler does the dynamic portion (as we already compute it) of the cons/car/cdr in the shape environment. For car/cdr, if the value in the shape env is atomic, then a dynamic instruction is produced.
7. When the compiler generates a known procedure call, the shapes of the dynamic arguments are known, so they can be split. If the call is inlined then the environment can simply be renamed.

The root of this problem is in the binding time lattice. At cogen time the length of a list is not known; circular binding times forget exactly this. But once the static values have been received, this information can be recovered.

While the above plan looks rather *ad hoc*, I believe that it could be much more simply implemented with a self-applicable specializer by using a binding time lattice that captures as static information what we put into the shape environment above. That is, $\widetilde{cons}$ `D D = (cons D D)`, etc.

[what is the state of the art in variable splitting for other systems?]

8.6 Lift Compiler

At various times a join in the abstract interpretation of the binding times loses information. This causes a variable to be lifted from one binding time to another. Simple cases are handled directly by cogen, but with partially static and circular binding times, lifting can be much more complex. This section describes how cogen uses itself to create *lift compilers* to handle complex lifting.

Some lifts are handled directly by cogen; these are *primitive* lifts. Eg, a lift from `S` to `D` causes a constant declaration to be generated by the compiler; the value of the constant is the static value, held in a local variable in the generated compiler. There is also the primitive higher-order lift, described below.

Consider the compiler for lifting `(cons (cons S D) (cons S D))` to `D`. In lisp, the compiler would look like [note compile]

```
(lambda (lift-bt-comp s-vals)
  (compile `(lambda (d-vals)
             (cons (cons ,(car s-vals) (car d-vals))
                   (cons ,(cdr s-vals) (cdr d-vals))))))
```

This example belies how hard lifting is in general: circular binding times, variable splitting, and higher-order lifting all make this harder. Fortunately, there is a nice way to handle all of this by using cogen itself. First define lift inductively (ignore higher-order procedures for a moment):

```
(define (lift bt-from bt-to val)
  (match (list bt-from bt-to)
         (('static 'dynamic)
          (primitive-lift val))
         ((x 'static) val)
         (('dynamic x) val)
         (else (if (pair? val)
                   (let ((a (lift (bt-car bt-from)
                                  (bt-car bt-to)
                                  (car val)))
                         (d (lift (bt-cdr bt-from)
                                  (bt-cdr bt-to)
                                  (cdr val))))
                     (cons a d))
                   val))))
(define bt-car ...) ; = car̃
(define bt-cdr ...) ; = cdr̃
```

[note match]

This procedure copies `val` while traversing the binding times in parallel fashion. When the binding times indicate that we have reached a part of `val` that must be lifted, then the primitive lift is applied. If we execute this procedure, it would only copy the value. Instead, use the *lift compiler*

```
(code lift (bt-from bt-to val k) ...)
```

```
(define lift-compiler (cogen lift '(static static dynamic dynamic)))
```

to create a procedure specialized to particular binding times:

```
; lifter =
(define lift-bt (eval-root lift-compiler (list tlc from to)))
```

This procedure lifts the right parts of `val`, but the control flow based on the binding-times has been eliminated. Control flow based on `val` will remain because of the `pair?` in the recursive case. But it's still just a copier.

At the point of the lift, a call to this procedure is inserted into the source instruction list. `cont` finishes the intructions list, it takes the lifted value and the live variables as arguments. So when cogen reaches this procedure call, another compiler is generated:

```
(define cl-bt (cons (const cont) live-vars-bt))
(define lift-bt-comp (cogen lift-bt (list bt-from cl-bt)))
```

This is exactly what we had above. It's interesting how the static data remaining in `bt-from` is passed to `cont`. The control flow from the `pair?` calls remains only where there was a loop in the binding times (see spines) [how does this relate to memoizing compilers only where required?]

Higher-order lifts can be handled by adding an addition case to `lift` that invokes cogen's primitive higher-order lift. Also when it conses up the result, it must use `closure-cons` as appropriate.

So when cogen encounters a non-primitive lift, it inserts a procedure call to a specialized version of `lift` into the source code, and cogens this instead. The lift compiler is a good example of three (it will be four when the general lift is written in mini-scheme instead of root) layer composition (see layers).

Lifting a atom to a partially static binding time requires replicating the atom in both binding times (think of splitting an environment into two lists, each has its own terminating nil). This atom must be copied.

[note lift-compiler-cruft]

I think the lift compiler might simplify things enough that it could be used to support direct implementation of a multistage cogen, instead of using multipass.

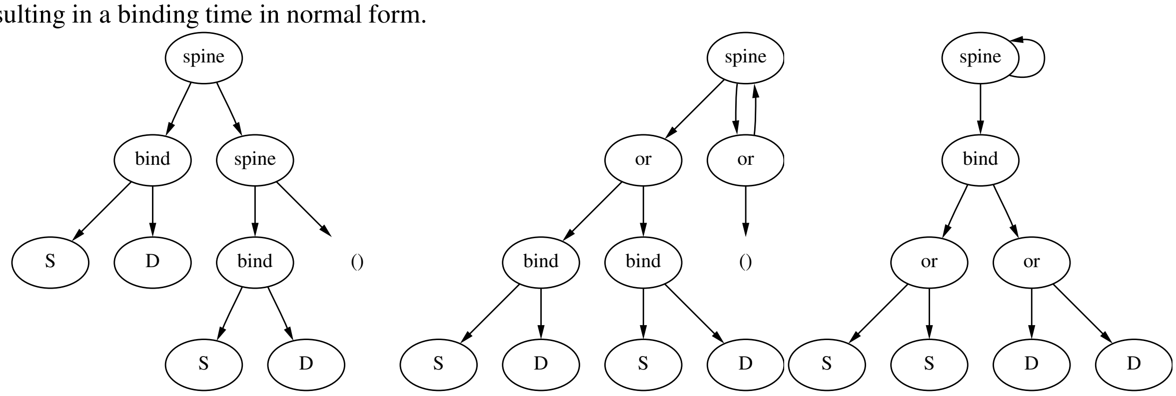
8.7 Eg Environments

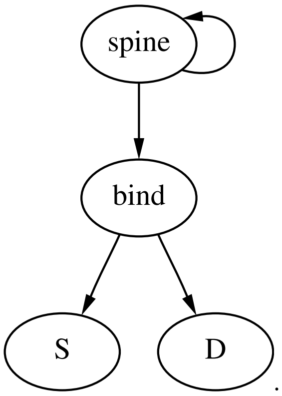
Without circular binding times cogen would often try to generate an infinite compiler. This section uses lexical environments to demonstrate how widening and memoization on binding times are used to replace this kind of non-termination with a loop in the compiler. It serves as a running example of how and why `cogen` works the way it does.

Consider generating a compiler for `alist-eval`, an interpreter that uses association lists (a list of pairs of names and values) to represent its environments. This interpreter's `apply` procedure calls `zip` (see zip) to build an alist, causing cogen to process the following non-terminating sequence:

```
; (code alist-eval (exp env k) ...)
; (code zip (vars vals r k) ...)

(zip S D (const nil) D)
(zip S D (cons (cons S D bind)
               (const nil) spine) D)
(zip S D (cons (cons S D bind)
               (cons (cons S D bind)
                     (const nil) spine) spine) D)
...
```

Without grammars, cogen would be forced to lift the third argument to `D`, but how could a compiler work without the names of local variables? The solution is to build circular binding times by collapsing duplicate nodes. These four diagrams show how the join nodes (labeled 'or') are successively reduced, resulting in a binding time in normal form.





.

But how does the loop in this binding time become a loop in the compiler? The answer: cogen memoizes on binding time patterns. Each `code` structure contains a memo table mapping binding time patterns to generating extensions. So the above sequence becomes:

```
(zip S D (const nil) D)
(zip S D (cons (cons S D bind) (const nil) spine) D)
(zip S D (cons (cons S D bind) self spine) D)
(zip S D (cons (cons S D bind) self spine) D)
```

In the last line, cogen finds the needed extension in the memo table of `zip`. The extension isn't complete yet, but it exists in the table; a side-effect is used on completion [note self-app-mutation]. Thus the compiler calls itself.

Note that the loop was only found after two iterations were 'unrolled'. This is a good example of excessive specialization (though there are cases where splitting off the first iteration of a loop is just what you want). Since the prototype is just a memoized forward-flow abstract interpretation, this is unavoidable. Doing better requires reverse information flow, eg a two pass BTA/generation system.

Now, think how this BT affects the compiler produced by cogen for `alist-eval`. In the compiler the environment is a list of variable names. That means the residual part of an environment is a list of variable values. Thus the code produced by such a compiler would contain only procedures called by passing fixed length lists. Each of these procedures is a residual version of `alist-eval`, that list argument is the `env` argument. A variable reference implemented with a call to `assoc` and a statically known variable would compile to a fixed (at compile time) sequence of `cdr`s followed by a `car`. This is exactly the inefficiency variable splitting is designed to alleviate (see splitting).

Note how this interacts with higher-order calls to unknown procedures, and handling of Scheme's varargs in general.

9 Methodology

This section summarizes the hoped-for contributions and my plan for achieving them. Subsections present the open questions, experiments, preliminary results, and research schedule in greater detail.

There are a number of questions that I hope to answer. I divide them into two categories: what are the analytical properties of the transformed code and its execution, and how practical are the transformations. The former resemble typical compiler performance evaluation questions, the latter are fuzzy software engineering issues (the 2nd Futamura projection has a human factor).

- Is the code generated by these compilers good enough? How does the overhead compare with the speedup? Is the root language too low level? Should it have a type system?

- Can *needed* compilers be generated from interpreters? Is the system too brittle? How do generated compilers compare to hand-written ones? Are binding times easier to use than fancy lisp macros? Does any of this really work at all??

My plan to answer these questions is to

1. implement the system
2. perform experiments
3. iterate system design and experiments based on the results
4. seek answers to the questions.

I will start with the following experiments (listed with keywords)

loop language general loops, side effects

protocol kit KMP searching, regexps, un/parsing, format

2D graphics memory layout, neighborhoods, bit-fields

mini-scheme letrec, exceptions, modules, varargs, towers

but as I iterate the design, I will cull some experiments to concentrate on the most promising subset. Preliminary results are presented in a following section.

As this is a inter-disciplinary thesis roughly between partial evaluation (PE) and interactive graphics, we can make three categories of the contributions:

strictly PE the lift compiler, CPS-CPS and higher-order values, formalization of variable splitting as abstract interpretation, understanding the return to self-application.

the combination loop language, bitmap layout language, synchronous dataflow language, perhaps more.

strictly graphics are high-performance graphics routines worth the effort?

9.1 Questions

Is the final code good enough? DCG assembles its somewhat higher level IR into code locally comparable to gcc's [GCC], using only about 350 instructions per instruction produced. Using a lower level representation allows more optimizations to be handled by generated compilers, but the lack of high level constructs complicates cogen since it can not use these same constructs. I don't plan on generating machine code so I will simply examine the generated abstract code and assess how much optimization will be required to produce fast code, and how quickly naive code could be produced. The root language may have to be extended to make efficient code generation easier, eg loops may require direct support.

Does the speedup justify the time spent compiling? Of course it depends on the application. I will measure abstract instruction counts to compare interpretation to compilation and execution, and determine the break-even point. This ignores scheduling, cache, and memory effects. I hope to show that my generated compilers can be significantly faster than typical lisp compilers and still produce good code.

Is the root language too low level? Some language features may not be easily handled once they are compiled into the IR, since information is lost by translation into a lower level language (eg short-circuit con/disjunctions, types, pattern matching, loops, letrec, varargs, higher-orderness, procedure call/return, etc). I will find ways to handle as many of these as possible. Some of them may require special features, new hints, or some generalization. Some may have to be included directly in the IR. This is a familiar drawback to an IR: it can handle many different languages, but none of them perfectly.

It may turn out to be helpful to impose a type system on the IR. This would at least obviate the `closure-cons` hint, any kind of side-effect purity hint, and probably clean up some dark corners in the soundness of the system.

Can the compilers that one wants be generated from interpreters? In a trivial sense this is always true, since we can always cook up an interpreter that arbitrarily transforms its program argument, but it's not necessarily usefully true: is cogen saving programmer time? The experiments provide a context for what 'one wants'.

Successful compiler generation depends on the exact results of complex analyses. An innocuous code change may result in a critical value being lifted, ruining the dynamic code. This is brittleness. Typical examples are currying a procedure and inserting a `let` binding. Working with code while maintaining good binding time division may simply be too difficult.

How do the generated compilers compare to hand-written ones? Hand-coded compilers are impossible to beat as there will always be global invariants undiscovered by automatic means. But `cogen` should be able to produce compilers that are globally not stupid and are locally tight. I will hand-write compilers for some of the languages, and compare them to the generated compilers.

Will `cogen` be just as complicated as advanced macro hacking in Lisp? Binding times attempt to alleviate `` `@,' `` ridden code, but introduce their own complications. I will make a cursory comparison to C's macro languages: `cpp`, `bison`, `make`, and the occasional `gawk` script...

These are contradictory goals. Current techniques do not completely automate compiler generation (and it's hard to imagine how they could). As one approaches full automation, predictability goes down as more and sophisticated analyses and inferences are required. Finding good trade-offs between competing goals is the hallmark of engineering.

9.2 Schedule

Estimated tasks and weeks, roughly in order:

```
splitting        3
inlining         3
lifting          3
loops            2
meta-writing     4
bta              3
writing i        4
protocols        3
graphics         4
cleaning         2
tower            1
scheme           2
analysis         4
writing ii       8
polishing        4
slack            4
writing iii      4
-------        ---
total           58
```

resulting in graduation in mid 1996.

10 Preliminary Results

The following sections describe the experimental results obtained so far:

loop language general loops, side effects

protocol kit KMP searching, regexps, un/parsing, format (some goals only, no code yet)

2D graphics memory layout, neighborhoods, bit-fields

mini-scheme letrec, exceptions, modules, varargs, towers, macros (nothing here yet)

10.1 Loop Language

Existing C vector libraries such as CVL [CVL] support high-performance, aggregate data manipulation and is intended as a intermediate language for [NESL]. This section outlines a loop language for uniprocessors that could run faster yet provide greater flexibility.

The bulk of CVL's procedures may be classified according to four axes (listed with their possible values):

primop plus, times, sqrt, etc.

type bit, int, double, etc.

pattern element-wise (elwise), scan, reduce.

representation segmented, flat.

These near-symmetries are also present: vector/scalar arguments, in-place operations, permutations/scatter/gather.

Because each procedure is written separately, the number of procedures in the library is the *product* of these options. The implementation is not (and could not be, due to the interface) blocked (see buffering), thus performance is limited by memory rather than cache bandwidth.

The idea behind the loop language is to use fewer, more general loops and specialize them. The compilers are loop generators where the operation (`op`) is passed in as a scalar procedure.

Here is a simple example:

```
;;; spec ⟶ (spec start stop step)
(define (linear-multi-loop op loop-specs init-state here)
  (if (null? starts)
      (op init-state here)
    (let ((spec (car loop-specs)))
      (let loop ((i (start spec))
                 (state init-state))
           (if (< i (stop spec))
               (loop (+ i (step spec))
                     (linear-multi-loop op (cdr loop-specs)
                                       state (cons i here)))
            state)))))
```

CVL's primop and pattern axes are now actual parameters of an 'interpreter'. Hand analysis indicates that applying `cogen` would produce an efficient compiler that produces basically efficient code.

Higher-order operations (even for scans and reduces) are handled by the loop since `op` need not be a primop. There's no way to handle higher-order scans and reduces without RTCG. [cite?!]

In addition, general multidimensional data is handled very simply; without a real metalanguage it presents a substantial challenge.

It's not hard to modify this general loop so that the generated compiler would unroll (say four times) the inner loop over the data (the loop over the dimensions is completely removed). This depends on the linearity of the loop to remove the tests; we're not just inlining to remove jumps. Also straightforward to handle are scalar/vector arguments, higher-order scatter/gather/permutations, and perhaps also sorting the dimensions for better memory access patterns.

Handling packed datatypes of sub-word sizes and segmented representations is much more difficult since code must be reorganized so data is transferred in fixed sized (see graphics).

Now, once one has this loop language, how does one build NESL on top of it? One might hope that writing a NESL interpreter using the loop language and then applying `cogen` would work, but there's no way avoid implementing something like [ChaBleFi91] if you want good performance. How can we compromise?

10.2 Protocol Kit

Consider string matching, regular expression matching (or lexing), and parsing. Traditional techniques involve compiling the language (string, regexp, grammar) into tables, then running an abstract machine driven by these tables. RTCG can be used to compile these tables (programs for the abstract machines) into executable code, resulting in faster execution.

In fact, PE has been shown to be sometimes able to generate the tables themselves for KMP string matching [?] and parsing [?].

Formatting (unparsing) is another important operation. In C it is done with `printf`, in lisp with `format`. The languages defined by these format strings can be quite powerful, and it makes sense to compile them. Eg say you want to print a number with some surrounding text. It is easier to say `(format #t "good  d bad %" num)` than

```
(write-string "good ")
(write-decimal num)
(write-string " bad\n")
```

but the latter runs faster. This is a familiar programmer's dilemma. By compiling the format string, you can have the best of both worlds.

If these sorts of procedures are implemented as stream processors, then they can be further composed. [FOX]

Ideal protocol usage is measured against Kolmogorov complexity [ref], which requires transmitting programs.

> The advent of high bandwidth networks, and an increased range of applications from video and voice to more traditional data transfer, is likely to show increasingly the inadequacies of the layered communications architecture as a model for constructing real systems. [CroWaWaSi92]

[corollary to the end-to-end principle?]

10.3 2D Graphics

This section presents preliminary results towards discovering how PE and RTCG can make the bitmap operations of 2D graphics code more modular and flexible, and sometimes faster too. First, the bitmap data-structure and some of its common operations are introduced. The implementation of two of these operations (bitblit and dithering) is then sketched, and explored further in two subsections. The difficulties of dithering are compounded in area-operations such as filtering. I suggest synchronous real-time languages as a promising source of techniques for efficiently compiling complex data-flow programs with partial evaluation. Finally we look at the display of an MPEG2 stream.

1. bcopy The Trick

2. dither partially static bit-fields

What is 2D graphics? 2D graphics is primarily concerned with the representation and manipulation of bitmapped images. [a bitmap is a two dimensional array of pixels where the boundaries may fall between words. there may be any number of bits per pixel, or they may be indexed. may have any number of channels, what's a channel. the most common kinds of bitmaps] [manipulation: per-pixel operations such as posterization and gamma correction, area operations such as filtering and cellular automata; blitting, scan-conversion (drawing lines and circles).

Because images are 2D arrays at heart, the loop language presented in section 10.1 is directly applicable to performing per-pixel operations on bitmaps.

Bitblit (short for 'bit block transfer') is a fundamental operation used for everything from scrolling a window to printing text. The classic example of RTCG is Pike's implementation of bitblit [blit]. Section 10.3.1 presents preliminary results showing how to implement bcopy (a special case of bitblit) with `cogen`.

Dithering is the operation used to display a 24-bit image on a display with only a few bits. A much more complex and sketchy description a general but fast medium-quality dither procedure is here. The ditherer handles interleaved/sequential channels in any order (but still shares loads and base addresses), dithers down to any number of bits in each channel. The ditherer is built from the following components: 1D linear byte-looper, software cache, 1D-feedback dither, byte permuter. The ditherer is specialized to the memory layout of its arguments.

Efficient implementations of area operations (eg CA and filtering, or error-diffusion dithering) use a sliding window to reduce memory references. If the area is 3x3, then a 3x4 window can reduce memory references (not cache misses) significantly ($\frac{4}{2}/9 = 0.22$). Similar operations include 2x2 decimation, pixel replication, tiling, and mirroring. To avoid shuffling data inside these loops, they can be unrolled once for each column of the window. [need a diagram] (4 scan lines * 32 bits/pixel * 1k pixels = 16k)

I think such problems may be solved with synchronous streams. The connection of the streams is static, but the data flowing between them is dynamic. Proper scheduling involves repeating loops of different lengths until they sync up, this can be handled by the cogen's memoization if the right values are made static. Cogen may be handle something like an interpreter for a synchronous real-time language like [StateCharts] or [signal].

The above discussion focused on computations where data is read exactly once, and temporary storage can be avoided by dynamically building loops. but sometimes it cannot be avoided: for example if you compose two filters, or when you multiply two matrices. Here's where blocking and tiling come in.

RTCG can also be used to simplify the implementation of prims like line drawing. Bresenham's algorithm's 8-way symmetry may be generated. [Sproull] shows how a number of efficient variations of the algorithm may be generated by code transformations, perhaps some of them can be expressed with cogen.

The caches and byte-memops instructions available in general purpose CPUs take care of most of the computation saved by making the shifts and masks static in the bcopy and dither examples. However, DSP chips (plus the Alpha [Alpha]) sometimes have no such features, and sometimes do provide explicit access to on-chip ram, sram, and dram.

An important example: fast decompression and display of a video stream. MPEG2 expands to full-color, but we want to display it on a cheap 8-bit display. Pixels come in 8x8 blocks which must be dithered then moved to the frame buffer. In the 2.5D and 3D realms, the pixels are projected and clipped in what is usually another layer of processing. Isn't this something like `prman`'s CAT (coherent access texture) feature? Looking 'end to end' (network device to frame buffer), how much opportunity for `cogen` is there? How close to the 'minimum' number of layers can `cogen` go, without using an unnatural means of organization to get there?

10.3.1 Bcopy and The Trick

This section explores how a typical bcopy optimization can be implemented with The Trick. The functioning of The Trick is explained in detail (the details are peculiar to my system, the concepts are not). Bcopy (copy a byte-addressed block of memory) is a simple kind of bitblit. Efficient implementations copy the memory word-at-a-time, but if the blocks' alignments differ then the data must be masked and shuffled in the registers. Roughly (ignoring the boundaries):

```
(define (bcopy from to nbytes)
  (let* ((diff (mod (- from to) 4))
         (diff (trick-n diff 4)))
    (bcopy1 from to nbytes diff)))

(define (bcopy1 from to nbytes diff)
  (let ((nwords (truncate (/ nbytes 4))))
    (define (shuffle src prev)
      (let* ((bits (* 8 diff))
             (mask (- (shift-left 1 (- 32 bits)) 1)))
        (bitwise-or (shift-left (bitwise-and mask src) bits) prev)))
    (let loop ((i 0) (prev 0))
         (let* ((s (vector-ref from i))
                (d (shuffle s p)))
           (vector-set! to i s)
           (loop (+ 1 i) (hibits s diff))))))
(define (trick-n x n)
  (cond ((< 0 n) x)
        ((= x n) n)
        (else (trick-n x (- n 1)))))
```

Cogen applied to this code with binding time pattern `(D D D)` does the right thing: even though all the parameters are dynamic the `trick-n` procedure produces four copies of the main loop, each specialized to a different alignment. The bit-shuffling is static. If needed one could also specialize to the length (eg when it's small).

The `trick-n` procedure implements the classic 'dynamic choice of finitely static values' PE trick [JoGoSe93]. This implementation produces a dynamic, linear chain of if-tests. A fancier version could produce a log-tree or a jump table. If the alignment is guaranteed, one may avoid the tests completely by specializing `bcopy1`.

Consider how a purely dynamic compiler functions like a macro. This works because of The Trick: converting a dynamic value into a finitely static one via case analysis. The finite (aka terminating) static computation is executed by the compiler (analogously to macro execution).

`trick-n` in `root` is

```
(code trick-n
      (x n k)
      ((const zero 0)
       (prim z>n ,> zero n)
       (if z>n (,@(call 'k '(x))))
       (prim nn ,identity n) ; prevent n from being lifted
       (prim n=x ,= nn x)
       (if n=x (,@(call 'k '(n))))
       (const one 1)
       (prim n-1 ,- n one)
       (const trick-n trick-n)
       (jump trick-n x n-1 k)))
```

Now instead of `bcopy`, consider the simpler `trick-n-demo` procedure. It retains the essential features: static computation depending on a dynamic parameter of statically known range. Note that in the bcopy example `max` is not a parameter, thus this example is somewhat more general.

```
(define (trick-n-demo actual max s d)
  (let ((a (trick actual max)))
    (* d (* s a))))
```

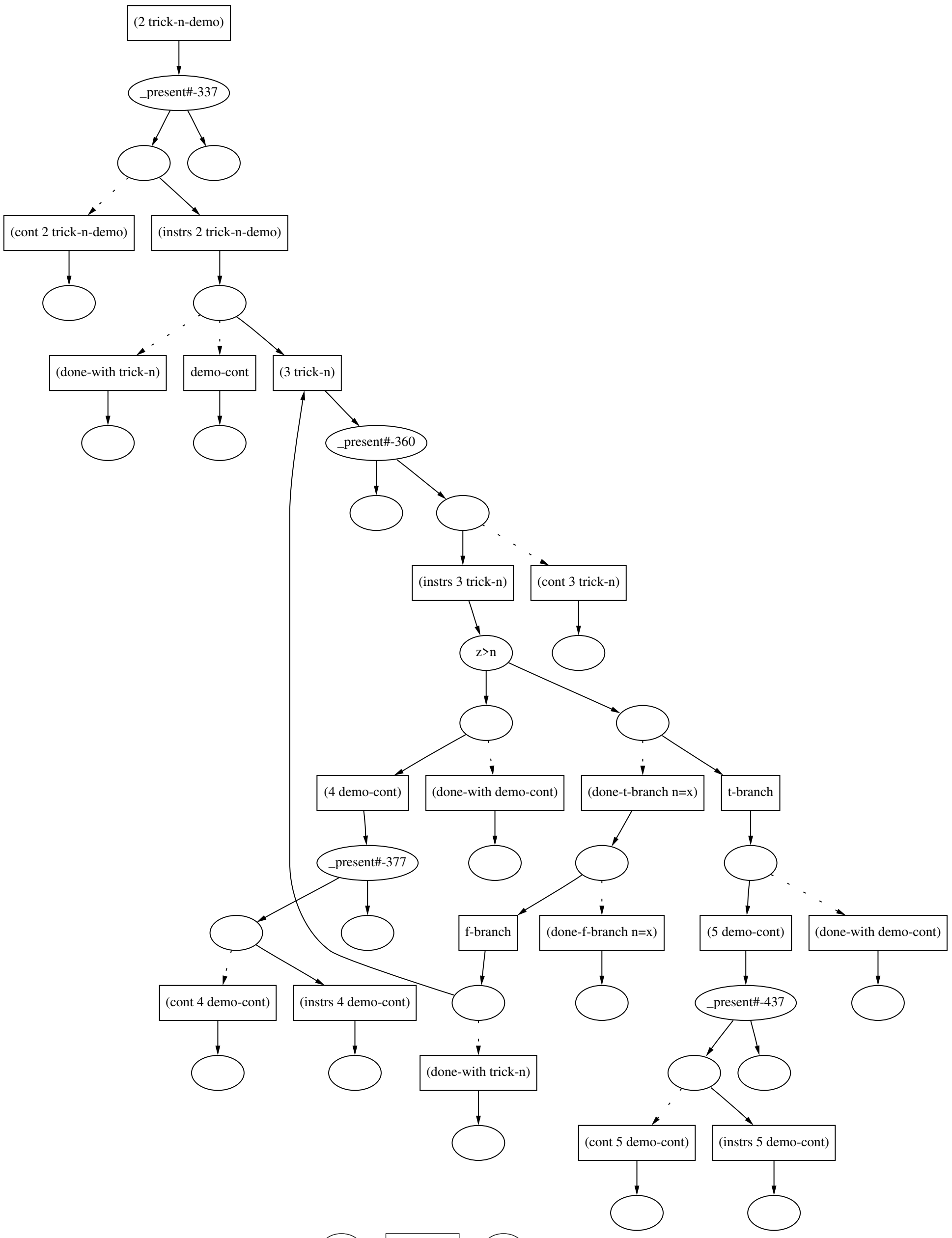
Even though `actual` is dynamic we can eliminate the inner multiplication provided that it is less than `max` (in bcopy, this corresponds to the offsets for the masking and shuffling in the loop). In `root` this is

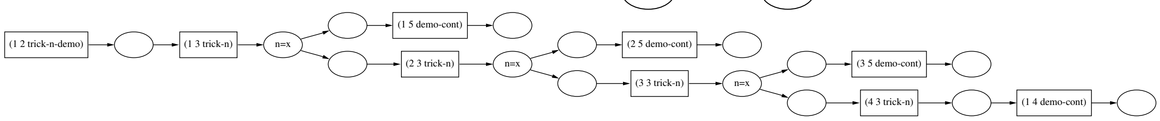
```
(code trick-n-demo
      (actual max s d k)
      (,@(make-closure 'demo-cont '(s d k) 'cl)
       (const trick-n ,trick-n)
       (jump trick-n actual max cl)))

(code demo-cont
      (cl arg)
      (,@(unmake-closure '(s d k) 'cl)
       (prim r3 ,* arg s)
       (prim r4 ,* r3 d)
       ,@(call 'k '(r4))))
```

We can generate versions of `demo-cont` specialized to values for `actual` from 0 to 2 and also to the rest of the static environment (s $\rightarrow$ 7).

```
(define tndc (cogen trick-n-demo '(dynamic static static dynamic dynamic)))
(define fast-demo (eval-root tndc (list top-cont 2 7)))
```

The code for `fast-demo` includes 3 copies of `demo-cont`. In each, the first multiplication (`arg` with `s`) has been performed. See how the loop in the compiler `tndc` expands in its output `fast-demo`.





`fast-demo` also includes a general version of `demo-cont`, in case the actual value were greater than `max`. If this is not needed, then a version of `trick-n` that produces an error instead of returning `x` could be used. Even in cases where the test against zero will never return true, the test cannot be eliminated because that produces a non-terminating compiler. Though this is a good example of how the programmer must understand and account for PE in their code, note that the accounting is limited to the `trick-n` procedure, rather than `demo-cont`.

10.3.2 Dithering and Per-pixel Ops

Dithering makes a good example of how `cogen` can make an abstract implementation run fast. This section sketches a procedure that dithers a 24-bit rgb image down to any size color-cube. The rgb image's channels may have any linear organization. A software cache converts loop code with byte-loads into code with word loads, shifts, and masks. It works by keeping the low two bits of a loop index static. The ditherer is built from the following components: a 1D linear looper, the software cache, a hash-noise dither, and a byte permuter.

Why RTCG: The size of the destination color-cube depends on how many unallocated colors are available in the X server, and thus isn't available until runtime. Furthermore the user may want to display images from different sources with different channel layouts.

In this section, the partially-static integers and the operations on them are manually. This is a new [i've never seen it, have you?] form of binding time improvement. It seems likely that applying the idea behind partially static structures to partially static bit-fields could automate this division. Eventually an analogue of variable splitting might be used to further optimize a number of bit-preserving operations.

If `x` is a dynamic variable, then `x2` is the static low two bits.

```
; index and offset.  one offset per channel.
loop((i, i2), ((o, o2) ...)) =
  if (i2 == max2 and
      i == max)
    return;
  loop
    b = byte-load-cached(heap-p, o, o2);
  f(b, ...);
  loop(plus((i,i2),(0,1)), (plus((o,o2),step2) ...))

plus((a, a2), (b, b2)) =
  e = a2 + b2;
  f = a + b;
  if (e > m)
    (f + 1, e - m)
  else
    (f, e)

; state: (p0, wo0, w0)
; records previous heap pointer, word offset, byte offset
; when is the state initialized???
byte-load-cached(p, wo, bo) =
  if (p = p0 and wo = wo0)
      w = w0
    else
      w0 = w = word-load(p, wo)
      wo0 = wo
      p0 = p
  b = byte(w, bo)
```

Elimination of the cache test requires variable splitting and improving the binding time of the `eq?` function so that `(eq? v v)` → `#t`.

When the compiler runs it could produce eg:

```
loop(i, or, og, ob) =
  w = word-load(heap-p, or);
  r = byte(w, 0);
  g = byte(w, 1);
  b = byte(w, 2);
  f(r, g, b);
  r = byte(w, 3);
  w = word-load(heap-p, or);
  g = byte(w, 0);
  b = byte(w, 1);
  f(r, g, b);
  r = byte(w, 2);
  g = byte(w, 3);
  w = word-load(heap-p, or);
  b = byte(w, 0);
  f(r, g, b);
  r = byte(w, 1);
  g = byte(w, 2);
  b = byte(w, 3);
  f(r, g, b);
  if (i == max) return;

  loop(i+1, o+1);
```

Similar optimizations include allowing the user to write a one-channel filter function, and then applying it to grayscale, rgb, and rgba images without loss of performance.

11 Follow-up Work

No doubt many of the happy plans of the previous sections will remain 'to do' even after the thesis is complete. The runtime/backend system contains many opportunities for improvement and extension. Alternatively, techniques developed with nitrous could be grafted onto an existing system. This section lays out directions for future work.

11.1 Backend

First, the system currently only produces intermediate code. I have a jury-rigged root→scheme compiler working in the scheme48 system, which compiles scheme to interpreted bytecode. To get any kind of real performance measurements, a custom root→machine-code backend is required. This involves primarily register allocation, instruction scheduling, jump optimization, and dead code elimination.

After a simple backend is working, more powerful global optimizations can be incorporated, using feedback from execution. Take ideas from Self [self] and code coagulation[ref]. Side effects can be isolated by using three levels of purity: pure (inline freely), infrequently changed (propagate, but keep back-pointers for when the value is mutated), and volatile (do not inline/propagate). This ties into the code developement system, and how a programmer changes the definition of a value/procedure on the fly.

Current CPU architectures have fixed maximum instruction level parallelism because they are implemented with fixed CMOS gates; PE sometimes produces extremely wide code, wider than any imaginable general purpose CPU. Field Programmable Gate Arrays (FPGAs) are another new chip technology, they allow their gates to be reconfigured 'at run time', and thus provide an interesting target for a future backend [IseSa93].

11.2 Cogen

Cogen itself has many deficiencies, even after this thesis is complete. The following are possible courses of action:

- handle general records rather than just pairs. OOP?
- handle side effects in compilers [Heintze94].
- extend the analysis to track eq-ness of values and their purity.
- convert fixed-length lists into records (unboxing according to static structure (see spines)).
- switch to conditional, procedure-call, exceptions, and loop based control flow rather than CPS-CPS.
- return if possible to a self-applicable specializer (from a directly implemented compiler generator). This is much more elegant and contains less duplicated functionality, though it's not clear if it can be done without losing important features.
- extend it from two-stage to multi-stage.

11.3 Experiments

The proposed instruction count measurements could be improved by profiling the abstract code with progressively more accurate models of memory hierarchy and CPU resources. One way to implement such a model is with a `root` language self-interpreter that records execution statistics. For example, by modeling the instruction cache one could answer the question: does creating lots of specialized code blow the instruction cache, or is that offset by eliminating conditionals?

Complex, statistics-collection memory-model simulators are well served by RTCG. It may be possible to generate the simulators by writing `root` self-interpreters that collect statistics, then using them to compile the programs you want to benchmark. Similar techniques may provide safe languages for kernel environments. Hardware resource sharing of the clock (non-preemptive threads) and memory (GCed heaps) also make interesting targets for code transformation.

fecund experiment list: OOP (prototypes, first class environments), interpreter kit (monads, monadoids), fnord, diffey-q package (convert to finite-difference), interactive DSP programming, a network distributed paint program, a visual-musical instrument, artificial life, genetic art, structure browser/editor/outliner, janus, a lisp OS.

12 Bibliography

Abelson92 Hal Abelson's foreward to Essentials of Programming Languages; D P Friedman, M Wand, C T Haynes; MIT Press 1992. ISBN 0-262-06145-7 local.

Alpha Alpha Architecture Reference Manual; ed Richard L Sites; Digital Press. ISBN 1-55558-098-X.

Appel Compiling with Continuations; Andrew Appel; Cambridge University Press; ISBN 0-521-41695-7.

Apply An Architecture Independent Programming Language for Low-Level Vision; L G C Hamey, J A Webb, I-Chien Wu; Computer Vision, Graphics, and Image Processing 48, 1989.

BaGluXX Partial Evaluation of Numerical Programs in Fortran; R Baier, R Glück, R Zöchling; ????.

BiWe93 Partial Ealuation of Standard ML; Lars Birkedal, Morten Welinder; DIKU TR 93/22.

blit Hardware/Software Trade-offs for Bitmap Graphics on the Blit; Rob Pike, Bart Locanthi, John Reiser; Software-Practice and Experience, Vol 15(2), 131-151 (Feb 1985).

blond A Blond Primer; Olivier Danvy, Karoline Malmkjaer; DIKU TR 88/21.

BoDu93 Handwriting Cogen for a CPS-Based Partial Evaluator; Anders Bondorf, Dirk Dussart; ????93.

CAM Cellular Automata Machines; Toffoli, Margolus; MIT Press. ISBN 0-262-20060-0.

ChaBleFi91 Size and Access Inference for Data-Parallel Programs. Siddhartha Chatterjee, Guy E Blelloch, Allan L Fisher. SIGPLAN91.

Chez Cadence Research Systems; Bloomington Indiana; Chez Scheme. related paper by the same author. local.

CodeWarrior Metrowerks Inc; St. Laurent, Canada; CodeWarrior(tm).

Consel90 Binding Time Analysis for Higher Order Untyped Functional Languages; Charles Consel; LFP90.

Consel93 Polyvariant Binding-Time Analysis For Applicative Languages; Charles Consel; PEPM93.

Cook84 Shade Trees; R L Cook; SIGGRAPH84.

CroWaWaSi92 Layering considered harmful; Crowcroft, Wakeman, Wang, Sirovica; 1/92 IEEE Networks.

CVL CVL: A C Vector Library Manual, Version 2; Blelloch, Chatterjee, Hardwick, Reid-Miller, Sipelstein, Zagha; CMU-CS-93-114. local, original.

DCG DCG: An Efficient, Retargetable Dynamic Code Generation System; Englar, Proebsting; ASPLOS94; local, original.

Dragon Compilers: Principles, Techniques, and Tools; A V Aho, R Sethi, J D Ullman; Addison-Wesley 1986. ISBN 0-201-10088-6.

elisp GNU Emacs Manual; Richard Stallman; Free Software Foundation 1987. manual

Fabius Lightweight Run-Time Code Generation; Leone, Lee; PEPM94. local, original.

FraHa91 A Code Generation Interface for ANSI C; C W Fraser, D R Hanson; Software Practice and Experience 21(9) 1991.

Fuse Towards a New Perspective on Partial Evaluation; Morry Katz, Daniel Weise; PEPM92. local.

Fuse Graphs as an Intermediate Representation for Partial Evaluation; Weise; FUSE-MEMO-90-1; local, original archive.

GCC Using and Porting GNU CC; R M Stallman; Free Software Foundation 1989. manual.

GluJoXX Generating Optimizing Specializers; R Glück, J Jørgensen; ICCL94.

GoJo88 Compiler Generation by Partial Evaluation: a Case Study; Carsten Gomard, Neil Jones; DIKU TR 88/24. [good explanation of how self-application works via a flow-chart language].

GruZo92 CustoMalloc: Efficient Synthesized Memory Allocators; Dirk Grunwald, Benjamin Zorn; CU-CS-692-92.

GuSto82 A Language for Bitmap Manipulation; Leo Guibas, Jorge Stolfe; ACM Transactions on Graphics Vol 1 No 3 July 1982.

Heintze94 email 1994. local.

Henglein91 Efficient Type Inference for Higher-Order Binding-Time Analysis; Fritz Henglein; FPCA91.

HyperNeWS `ftp://ftp.win.tue.nl/pub/hypernews`

IseSa93 Spyder: A Reconfigurable VLIW Processor using FPGAs. Christian Iseli, Eduardo Sanchez. IEEE Workshop on FPGAs for Custom Computing Machines 1993

JoGoSe93 Partial Evaluation and Automatic Program Generation; N Jones, C K Gomard, P Sestoft; Prentice-Hall 1993. ISBN 0-13-020249-5.

Jorgensen92 Generating a Compiler for a Lazy Language by Partial Evaluation; Jesper Jørgensen; POPL92.

JoSche86 Compilers and Staging Transformations; Ulric Jørring, William Scherlis; POPL86.

KeEggeHe91 A Case for Runtime Code Generation; D Keppel, S J Eggers, R R Henry; UW-CSE-91-11-04. local, original.

KeEggHe93 Evaluating Runtime-Compiled Value-Specific Optimizations; D Keppel, S J Eggers, R R Henry; UW-CSE-91-11-04. local, original.

Kolmogorov An introduction to Kolmogorov complexity and its applications; M Li, P M B Vitányi; Addison-Wesley 1992.

LaKiRoRu92 An Architecture for an Open Compiler; J Lamping, G Kiczales, L Rodriguez, E Ruf; IMSA92 (Workshop on Reflection and Meta-level Architectures).

Lampsons-hints Hints for Computer System Design; Butler Lampson; SOSP83?.

Mogensen89 Binding Time Aspects of Partial Evaluation; Torben Mogensen; DIKU PhD thesis 89. TR#?

monads The essence of functional programming (Invited talk); Philip Wadler; POPL92.

MOP The Art of the Metaobject Protocol; Kiczales, des Rivieres, Bobrow; MIT Press. ISBN 0-262-61074-4.

NESL NESL: A nested data-parallel language version 2.6; Guy Blelloch, Siddhartha Chatterjee; CMU-CS-93-129. localoriginal.

On-Lisp On Lisp: Advanced Techniques for Common LISP; Paul Graham; Prentice-Hall 1994. ISBN 0-13-030552-9 local.

Peachey94 personal email 1994.

pseudomonads Building Interpreters by Composing Monads; G J Steele; POPL94.

PuWa93 A Study of Dynamic Optimization Techniques: Lessons and Directions in Kernel Design; Calton Pu, Jonathan Walpole; OGI-CSE-93-007.

r4rs Revised 4 Report on the Algorithmic Language Scheme; ed William Clinger, Jonathan Rees. local.

RenderMan The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics; Steve Upstill; Addison-Wesley. ISBN 0-201-50868-0.

SchemePaint Programmable Applications: Interpreter Meets Interface; Michael Eisenberg; MIT AI Memo 1325 1991.

Schism A Tour of Schism: A Partial Evaluation System For Higher-Order Applicative Languages; Charles Consel; ????

Schooler84 Partial Evaluation as a means of Language Extensibility; Richard Schooler; MIT-LCS-TR-324 1984. [online PE only]

self `http://self.stanford.edu`

SiBle91 Collection-Oriented Languages. Jay M Sipelstein, Buy E Blelloch. Proceedings of the IEEE, April 1991.

Signal Programing real time applications with Signal; P Le Guernic, M Le Borgne, T Gauthier, C Le Maire; Proceedings of the IEEE, Special Issue, Sept 1991.

Similix Automatic Autoprojection of Recursive Equations with Global Variables and Abstract Data Types; Anders Bondorf, Olivier Danvy. DIKU TR 90-4.

Similix-doc Similix 5.0 Manual; Bondorf. local.

Sims91 Artificial Evolution for Computer Graphics; Karl Sims; SIGGRAPH91.

SML/NJ Standard ML of New Jersey; A Appel, D B MacQueen; PLILP91. local, archive

SPIN SPIN—An Extensible Microkernel for Application-specific Operating System Services; Bershad, Chambers, Eggers, Maeda, McNamee, Pardyak, Savage, Sirer; UW-CSE-94-03-03. local, originalnexus.

synthesis Efficient Implementation of Fundamental Operating System Services; Henry Massalin; Columbia PhD thesis 1992.

Synthetix Incremental Partial Evaluation: The Key to High Performance, Modularity and Portability in Operating Systems; Consel, Pu, Walpole; PEPM93. local.

ThoDa95 Optimizing Software Synthesis Performance; Nicholas Thompson, Roger Dannenberg; submitted to 1995 International Computer Music Conference. local.

tick-C ‘C: A Language for High-Level, Efficient, and Machine-independent Dynamic Code Generation; Dawson R Engler, Wilson C Hsieh, M Frans Kaashoek; submitted to PLDI95. local, original

Visual-Basic Microsoft; Redmond Washington; Visual Basic v3.0 for Windows. original, in Excel, web nexus.

WiKa91 Reaction-Diffusion Textures; Andrew Witkin, Michael Kass; SIGGRAPH91.

WoLa91 A Data Locality Optimizing Algorithm. Michael Wolf, Monica Lam. SIGPLAN91.

YaSa Qua-C: Binary Optimization for Fast Runtime Code Generation in C. Curtis Yarvin, Adam Sah. local.

13 End Notes

coding We are passing the `power` function, but `cogen` needs to explicitly manipulate some coded representation of the procedure. I handle this by extending the run time representation of functions with their IR source.

compilation What is compilation anyway? There are bytecode compilers, compilers to threaded code, naive compilers to native code, and seriously optimizing compilers.

direct-multistage The hard part of a directly implemented multistage compiler generator appears to be lifting. Something like the lift compiler may be a good way to address this.

emacs-languages-all more sublanguages in emacs: printf format language, documentation string language, interactive argument prompt language...

higher-order-cruft Note this makes lifting non-transitive. Say instead you had lifted twice:

```
(cons (const cd) (cons ...) _ #t) →
(cons D (cons ...) _ #t) → D
```

In this case the information is lost. This is a potential source of brittleness, though I can't imagine how it could happen.

If you were also to handle the

```
(cons (const cd) (cons ...) _ #t) -> (cons S D)
```

case, then the car could be `S` instead of `D`. Is this useful?

lift-compiler-cruft The lift compiler is not solid yet; some cruft remains in its implementation (and thus in my understanding of the situation). Since this is an inductive definition of lift, when we are cogening it, we must not need to lift (or if we do it must be somehow simpler, leading to termination. But really, I think its best to avoid it entirely). This is the source of the `lift-if-circular` hints. Furthermore the `cons` in the recursive case should be marked as non-collapsing (haven't really confirmed this last one yet, and there's no mechanism for that kind of hint either (though they sure are easy to add)). Once this is working for sure, then I'll explain it it more detail. More importantly, this all should change for the simpler when real BTA becomes available.

match Andrew Wright's `match` macro performs pattern matching and destructuring. It's documentation is available here. I recommend it.

meta-static It's called meta-static analysis instead of just static analysis to avoid confusion with static values, which don't arrive until the compiler is executed.

name Nitrous oxide (N_2O) makes racing cars go fast, see here. Also here for a different perspective.

removing-times-one this compiler produces an unnecessary multiplication by 1.0, not present in the hand coded `power-20`. It can either be removed by the backend (here backend means either the lisp compiler or DCG), or an optimizing interpreter can be used (an interpreter that avoided the multiplication with an additional case).

run-time What is 'run time' anyway? It's generally a contrasting term with 'compile time'. The distinction is just an artifact of the barriers put up by the operating system and 'convention'. (Compare the price of calling a procedure on a buffer and and running a process on a file.) The whole point is to reduce the barrier to getting at that compiler.

self-app-mutation The side-effects used in cogen's memo-tables are exactly what interferes with a return to self-application (see future). [Heintze94] suggests side-effects can be handled. Since memo-tables are relatively well-behaved mutations, I expect a nice solution is possible.

shade-power If the surface and the light are both white, and the eye and the light are exactly lined up, then the shading procedure reduces to an power procedure. It's a generalization of b^e in the same way that cb^e is.

typed-interpreters One could recognize when a particular variable has function type, but if procedures are used to represent procedures, as one has to in order for all this to work, then this is equivalent to saying that the programs run by the interpreter don't have type errors. That would amount to a proof that the implementation of the interpreter's (hypothetical) type checker is correct, and this is practically out of reach for now, though conceivable with hand annotation to guide the proof.