The Anatomy of a Loop

A story of scope and control

Olin Shivers *

Georgia Institute of Technology

shivers@cc.gatech.edu

Abstract

Writing loops with tail-recursive function calls is the equivalent of writing them with goto's. Given that loop packages for Lisp-family languages have been around for over 20 years, it is striking that none have had much success in the Scheme world. I suggest the reason is that Scheme forces us to be precise about the scoping of the various variables introduced by our loop forms, something previous attempts to design ambitious loop forms have not managed to do.

I present the design of a loop package for Scheme with a well-defined and natural scoping rule, based on a notion of control dominance that generalizes the standard lexical-scope rule of the λ-calculus. The new construct is powerful, clear, modular and extensible.

The loop language is defined in terms of an underlying language for expressing control-flow graphs. This language itself has interesting properties as an intermediate representation.

Categories and Subject Descriptors D.3.3 [*Programming languages*]: Language Constructs and Features—control structures, frameworks, procedures, functions, and subroutines; F.3.3 [*Logics and meanings of programs*]: Studies of program constructs—control primitives and functional constructs; D.1.1 [*Programming techniques*]: Applicative (Functional) Programming; D.3.1 [*Programming languages*]: Formal Definitions and Theory—semantics and syntax

General Terms Design, Languages, Performance, Theory

Keywords Functional programming, control structures, iteration, lambda-calculus, compilers, programming languages, continuations, macros, Scheme

1. The case for loop forms

In call-by-value functional languages such as ML or Scheme, we typically write loops using tail-recursive function calls. This is actually a terrible way to express program iteration, and it's not hard to see why. As was popularised by Steele [14, 15, 16, 17], a tail call is essentially a "goto that passes arguments." So writing loops with tail calls is just writing them with gotos. Yet, it has long been accepted in the programming-language community that goto is a low-level and obfuscatory control operator, a position stated by Dijkstra's "Goto considered harmful" letter [5].

Consider, for example, the loop shown in Figure 1, which has non-trivial control structure. The version on the left is written using basic Scheme and tail calls, while the version on the right is written using a macro-based loop form. We loop over list `l`, binding elements of the list to variable `x`; the loop's final value is accumulated in `result` in reverse order. The body of the loop has two "fast path" tests. The first test allows us to skip the presumably expensive computations producing the intermediate values bound to `y` and `z`. The second test allows us to skip the `z` computation.

From a software-engineering perspective, this code fragment is a disaster. The fact that we are iterating over a list is expressed by code that is spread over the entire loop: the entry point binding `rest`, the four distinct calls to `loop` (including the initial call providing the beginning value `l` for `rest`), and the `null?` termination test. There's no clear distinction in the syntax between the loop parameter that *drives* the loop (`rest`), and the loop parameter that accumulates the loop's final value (`result`). Since both driver and accumulator are lists, if we slip up and swap the two update forms in one of our four tail calls, we won't get a type error (at run time in Scheme, or at compile time in the analogous SML variant), so the bug would be tricky to catch. As loop variables proliferate in more complex loops, and loop size increases, separating looping calls from the binding sites, this error becomes easier and easier to make. The interactions between the control structure of the fast-path tests and the environment structure imposed by lexical scope cause the loop's indentation to drift off to the right of the page.

The technology we'll be developing in this paper allows us to create cleaner sub-languages that can be embedded within Scheme that are specifically designed to express iteration. Using a loop form of this sort, we can write the same loop as shown on the right side of the figure. Even without going into the details of the various clauses (a grammar for the loop form is shown in Figure 3), it's fairly clear to see the high-level structure of the loop. It's clear that we are iterating over a list, that we have two fast-path tests, that we are binding `y` and `z` to the results of intermediate computations, and that we are accumulating the results into a list. The fact that the rightward-drifting/nested indentation of the original code has aligned into a vertical column is a hint that our control and environment structuring has been brought more closely into alignment.

The real payoff from specialised loop forms comes when we decide to alter the code. Suppose we wish to change our example so that the loop's input is provided in a vector instead of a list. Although this is conceptually a single element of the loop's structure, the relevant code that must be altered in the low-level tail-call version of the loop is spread across the entire loop, interwoven with the rest of the code. Figure 2 shows the new loop; you are invited

* This work supported by the NSF Science of Design program.

ICFP'05 September 26–28, 2005, Tallinn, Estonia.

```
(letrec
  ((loop (λ (rest result)
           (if (null? rest)
               (reverse result)
               (let ((x (car rest)))
                 (if (p? x)
                     (let ((y <verbose code using x>))
                       (if (q? <verbose code using y>)
                           (let ((z <verbose expression>))
                             (loop (cdr rest)
                                   (cons <z expression>
                                         result)))
                           (loop (cdr rest) result)))
                     (loop (cdr rest) result)))))))
  (loop l '()))
```

```
(loop (for x in l)
      (when (p? x))
      (bind (y <verbose code using x>))
      (when (q? <verbose code using y>))
      (bind (z <verbose expression>))
      (save <z expression>))
```

Figure 1. A loop expressed directly with tail calls and by means of a loop form. The loop form disentangles the distinct conceptual elements of the iteration.

```
(let ((len (vector-length v)))
  (letrec ((loop (λ (i result)
                   (if (= i len)
                       (reverse result)
                       (let ((x (vector-ref v i)))
                         (if (p? x)
                             (let ((y <verbose code using x>))
                               (if (q? <verbose code using y>)
                                   (let ((z <verbose expression>))
                                     (loop (+ i 1)
                                           (cons <z expression>
                                                 result)))
                                   (loop (+ i 1) result)))
                             (loop (+ i 1) result)))))))
    (loop 0 '())))
```

```
(loop (for x in-vector v)
      (when (p? x))
      (bind (y <verbose code using x>))
      (when (q? <verbose code using y>))
      (bind (z <verbose expression>))
      (save <z expression>))
```

Figure 2. Using tail calls, small design changes induce large code changes; with loop forms, small design changes require small code changes.

to hunt for the eight distinct changes scattered across the fifteen lines of code. If, however, we use a dedicated loop form, the single change to the loop's spec requires a single change to the loop's code. The loop-form version allows us to modularly construct the loop by separately specifying its conceptual elements.

When we increase the complexity of our loops, the problems associated with using direct tail calls to express iteration get worse. Multiple nested loops, "Knuth-style" exits occurring in the middle of an iteration, multiple iteration drivers or accumulators, and other real-world coding demands all increase the confusing complexity of this approach.

In short, tail calls and λ forms make great "machine code" for expressing a computation at a low level. But they are not good software-engineering tools for programmers to express an iteration.

Additionally, the problem is not handled by the use of higher-order functions such as `map` and `fold` to capture patterns of iteration. While these functional control abstractions can capture simple loops, they don't scale gracefully as loop complexity grows. For example, `map` is fine if we wish to accumulate a list... but not if some elements of the iteration don't add elements to the result (as is the case in our example above). We can use `fold` to iterate across a list... but what do we use when we wish to iterate across a list, *and* increment an index counter, *and* sequence through a companion vector in reverse order? Loop forms let us compose these loops easily by adding driver clauses in a modular fashion; functional abstractions do not compose in this way.

For these reasons, programmers in the Lisp family of languages have long resorted to loop-specific forms of the kind employed on the right-hand side of the examples above. Common Lisp, Zetalisp, and Maclisp all had loop forms more or less along the lines of the one we showed. Common Lisp's form is a particularly powerful and baroque example of the genre. It's notable, however, that this style of loop macro has never made much headway in the Scheme community.

2. The problem with loop macros

The critical problem with loop macros is the difficulty of providing a clear, well-defined definition of scope for the variables and expressions introduced by the clauses. A careful reading of the Common Lisp documentation for its loop form [18, paragraph 6.1.1.4] reveals the following somewhat embarrassing non-specification (bold-face emphasis added):

> Implementations can interleave the setting of initial values with the bindings. ... One implication of this interleaving is that it is **implementation-dependent** whether the *lexical environment* in which the initial value forms ... are evaluated includes only the loop variables preceding that form or includes more or all of the loop variables

In the commentary that accompanies the specification, Pitman's discussion [12] makes the ambiguity explicit (again, bold-face emphasis added):

> These **extremely vague phrases** don't really say much about the environment, and since they don't say what goes into the `let` or the `lambda`, or even how many `let` or

```
loop ::= (loop lclause ...)

lclause ::=
        (initial (vars init [step [test]]) ...)
      | (before exp ...)

      | (incr i from init [to final] [by step])
      | (decr i from init [to final] [by step])

      | (previous pvar var init_1 ... init_n)
      | (repeat n)

      | (for x in list [by step-fn])
      | (for l on list [by step-fn])
      | (for c in-string s [option ...])
      | (for x in-vector v [option ...])
      | (for i in-string-index s [option ...])
      | (for i in-vector-index v [option ...])
      | (for x in-file fname [reader reader])
      | (for x from-port exp [reader reader])
      | (for x input [reader])

      (continued)
      | bclause

      | (after exp ...)
      | (result exp_1 ... exp_n)
      | (save exp)

bclause ::=                                 ; Body clause:
        | (while  exp)                      ; Controls loop
        | (until  exp)                      ;   termination
        | (when   exp)                      ; Controls single
        | (unless exp)                      ;   iteration
        | (do exp_1 ...)                    ; For side effect
        | (bind (vars_1 exp_1) ...)
        | (subloop lclause ...)             ; Nested loop
        | (if exp bclause [bclause])

options ::= incr   | decr   | index i
        | from init   | to final   | by step
```

Figure 3. Partial grammar for the loop form—necessarily partial, as the clause set is extensible by means of the Scheme macro system.

`lambda` forms are involved, they **don't really say much at all**.

Further, the **vague statement** on p8–85 about how `let+setq` might be used to implement binding leaves an **unusually large amount of latitude** to implementations.

The source of the problem is that a single clause in a loop form can cause parts of the clause to be inserted into multiple places in the resulting code. For example, the `initial` clause in the loop form

```
(loop ...
      (initial (i 0 (+ i 1)))
      ...)
```

initialises loop variable `i` to 0, and then increments it on each following iteration. Thus, it introduces the expression `0` into the loop's prologue and the expression `(+ i 1)` into the loop's update section, which occurs at an entirely distinct position in the expanded code. This makes it problematic to define scope in terms of the order in which clauses occur in the loop form. If clause a occurs before clause b, a's late code fragment may come after b's early code fragment, but before b's late code fragment. Thus clause order cannot be used to establish some kind of scope order. Lack of a coherent, unambiguous scoping principle is not acceptable for a programming-language construct.

This difficulty in providing a clear specification for variable scope is, I believe, the primary reason Lisp-style loop forms have not been provided in Scheme. When we wish to make our loop form extensible, by allowing programmers to define their own clauses using Scheme's macro system, the problem becomes even more difficult.

3. The essence of lexical scope

The lexical-scoping rule of the λ-calculus provides two important properties for programmers. First, it allows a programmer to select a variable name for a local piece of code without needing global knowledge of the program. For example, if a programmer wishes to write a three-line loop somewhere in a program, he can select `i` as an iteration variable without having first to scan the entire program to ensure that `i` is used nowhere else. We can have multiple variables with the same name, locally disambiguating references to the shared name using the rule of lexical scope.

While important for humans, this property is not a "deep" property. Suppose we dispensed with it by defining a variant of the λ-calculus that required every variable declared in the program to be unique. This would cause no reduction in the computational power of the language; it would still be Turing-complete. A compiler or other program-manipulation system would have no problems analysing or otherwise operating on programs written in such a language.

The second property provided by classical lexical scoping, however, is a more generally useful one: definitions (or bindings) control-dominate uses (or references). That is, in a functional language based on the λ-calculus, we are assured that the only way program control can reach a variable reference is first to go through that variable's binding site. We have a simple static guarantee that program execution will never reference an unbound variable. If this seems simple and obvious, note that it is not true of assembler or Fortran: one can declare a variable in a Fortran program, then branch around the initialising assignment to the variable, and proceed to a reference to the variable. Further, if we are casting about for a general principle on which to base a language design, the simpler and more obvious, the better.

4. BDR scope

In the previous section, we established that the lexical-scope rule of the classic λ-calculus, which we'll call "LC scope," implies an important property: it ensures that binders dominate references. The key design idea that serves as the foundation for our extensible loop form is to invert cause and effect, taking the desired property *as our definition of scope*. This gives us a new scoping rule, which we call "BDR scope."

The BDR scoping principle can be informally defined in a graphical context. Suppose we have a control-flow graph. Each vertex represents a unit of computation, or "basic block;" there is a designated start vertex v_0; edges in the graph connect a vertex to its possible control successors. We additionally decorate each edge e with a set of identifiers vs_e. The total set of all identifiers occurring on the edges of the entire control-flow graph are called its *loop variables*. The execution model is that when control reaches a vertex, we perform the computation associated with that vertex. This computation culminates by (1) selecting one of the vertex's

```
cfg ::= (let*    ((l1 cfg1) ...) cfg)
      | (letrec  ((l1 cfg1) ...) cfg)
      | (go l)
      | (do proc (vars1 cfg1) ...)
      | (indep ((vars1 exp1) ...) cfg)
      | (permute      ((l1 cfg1) ...) cfg)
      | (permute/tail ((l1 cfg1) ...) cfg)
l ::= ident
vars ::= (ident ...)
proc, exp ::= Scheme expression
```

Figure 4. The CFG language

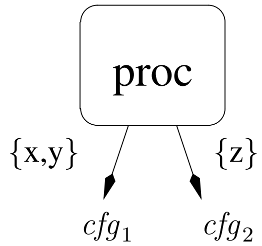


Figure 5. Individual vertex encoded as CFG `do` form.

out edges e, and (2) providing new values for the loop variables vs_e on edge e. We update these variables to the new values, and control proceeds along edge e to the next vertex. Note that this is a model that permits variable updating—that is, the value associated with a variable can change over time.

In our graphical model, the BDR scoping rule is this: vertex v is in the scope of loop variable x if every path from start vertex v_0 to v has some edge that defines x. If there is some way to get to v without defining x, then x is not visible at node v.

As we'll see, BDR scope is purpose driven for its intended application:

- It is chosen for application in an iterative, first-order context.
- It permits multiple updates to a given variable.
- It integrates with classic LC scope. That is, it allows us to specify the individual computations in the CFG's vertices with fragments of a standard, LC-scoped language, and then embed the entire CFG itself as a fragment within a larger program written in that LC-scoped language. This is precisely what we want for our intended use as an embedded Scheme macro.
- Scope is now an ensemble property of the entire control-flow graph. Thus we can stitch together fragments of a CFG from multiple sources (in our case, multiple clauses of a loop form), and then get a coherent scope story from the resulting graph.

In short, with the BDR rule, scope proceeds naturally from control.

5. The CFG language

The larger design picture is that we define our loop form by fixing a general iteration template; for our loop form, this template will have eight blocks (*e.g.*, an initialisation block, a per-iteration conditional top-guard block, a body block, an update block, a finalisation block, *etc.*). Each clause in a loop form will be a Scheme macro that expands into a small set of graph fragments, with each fragment targeted to one of the eight blocks in the general loop template. The loop form will collect the fragments from all the loop clauses, and insert them into the template, stitching them together into a complete loop, whose control and environment structure are given by its explicit control structure and its associated, control-determined BDR scoping rule. This graph is then translated by the macro into Scheme code.

The first step in this design is to make concrete a language for specifying control-flow graphs in composable fragments. We do this with the CFG language, shown in Figure 4. The CFG language has two independent name spaces, for binding *code labels* and *loop variables*:

- **Code labels**
  - name graph vertices;
  - are bound with `let*` and `letrec`,
  - referenced with `(go` l`)`,
  - and scoped with standard LC scope.
- **Loop variables**
  - name data values;
  - are defined on edge transitions,
  - referenced by the internal computations of the graph vertices,
  - and scoped using BDR scope.

Note that labels are not first-class, expressible values.

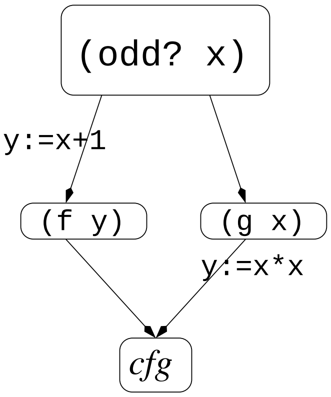
5.1 Specifying individual vertices with `do` forms

The basics of loop-variable handling are specified with the CFG `do` form, which is used to define a single vertex in the CFG and its outgoing edges. If the vertex has n outgoing edges, then edge i has attached loop-variable set $vars_i$ and leads to cfg_i. For example, Figure 5 shows, in graphical form, a `do` form that specifies a vertex with two exit edges. If the vertex's computation $proc$ chooses to exit along its first edge, it must provide two values which are used to update loop variables `x` and `y` before proceeding to cfg_1. If it exits along its second edge, it must provide a single value, which is used to update `z` before proceeding to cfg_2. The vertex's computation is expressed as a Scheme expression $proc$ (typically a `lambda` expression). When control transfers to this vertex, the $proc$ expression is evaluated in a Scheme lexical scope that "sees" the current bindings of all loop variables in the BDR scope of this vertex. This evaluation produces a Scheme procedure, which is applied to n procedures, each representing one of the vertex's n exit edges. When the computation represented by $proc$ is finished, it proceeds by tail-calling one of these exit-edge procedures, passing as parameters the new values for that edge's associated loop variables. It is an error to call an exit procedure in a non-tail position relative to $proc$'s application.[1] So, for example, if we want to encode a fragment of graph structure that updates `x` to be its absolute value and then jumps to the vertex named by graph label `next`, we can write the following CFG form:

```
(do (λ (e1 e2) (if (< x 0)
                   (e1 (- x))
                   (e2)))
    ((x) (go next))  ; Edge 1
    (( ) (go next))) ; Edge 2
```

This assumes that the `do` form executes in a loop-variable scope that includes `x`, and a label scope that includes label `next`.

[1] We could eliminate this errorful possibility simply by specifying the exit procedures to be real Scheme continuations produced by `call-with-current-continuation`. Conceptually, that is what they are: continuations. As a matter of engineering pragmatics, however, we can usually rely on a compiler to handle functional tail calls efficiently, while continuation creation and invocation remains a significantly heavier weight operation for many Scheme compilers. Since all this control structure happens behind the scenes, hidden by the surface syntax of loop-clause macros, it isn't visible to the application programmer in any event—it is solely the province of the meta-programmer who designs and implements the loop clauses.




```
(let* ((lj cfg) ; Join point
       (la (do (λ (e1) (f y) (e1))
               (() (go lj))))
       (lb (do (λ (e1) (g x) (e1 (* x x)))
               ((y) (go lj)))))

  (do (λ (e1 e2) (if (odd? x)
                     (e1 (+ x 1))
                     (e2)))
      ((y) (go la))
      (()  (go lb))))
```

Figure 6. A diamond control structure, rendered as a control-flow graph and as a term in the CFG language.

We specify a final or halting vertex in the control-flow graph simply by means of a `do` form that has no successor edges. In this case, the Scheme procedure halts the computation by returning instead of ending in a tail call. For example, if control ever reaches the vertex `(do (λ () 42))`, the CFG computation finishes, producing a final value of 42.

5.2 Composition: snapping together CFG fragments with label capture

The CFG `let*` form provides sequential lexical scoping for loop labels analogously to the Scheme `let*` form; this permits us to construct DAGs in our control-flow graph. Similarly, the `letrec` form allows circular binding of labels, so that we can construct graphs containing cycles. What's important about the particulars of this syntax is that we can use the LC scoping rule for loop labels to "wire up" fragments of graph structure. That is, if a CFG term represents a chunk of graph structure, then a free label—referred to by some internal `go` form but not bound by the term—expresses a "dangling" edge for the subgraph. If we embed the term within an outer CFG term that binds the free label, we provide a connection for the dangling edge.

For example, suppose we construct three arbitrary CFG terms, using the convention that each term jumps to a free label `p` (for "proceed") when it is done. Some CFG processor would like to wire the three terms together in series, so that when cfg_1 jumps to `p` with a `(go p)` sub-form, we transfer control to cfg_2; likewise, when cfg_2 jumps to `p`, we transfer to cfg_3. We do this by inserting the three fragments into the label-capturing template

```
(let* ((p cfg3)
       (p cfg2))
  cfg1)
```

As we'll see, this is how the fragments of control structure produced by the individual clauses of a loop form are snapped together into a complete loop.

5.3 Dominance trees and LC label scope

As a slightly more ambitious example, suppose we have a "diamond" control structure, as shown in Figure 6. The program tests loop variable `x`. If it is odd, we branch to the left child vertex (labelled `la` in the textual form), updating variable `y` to `x+1`. That vertex performs the Scheme call `(f y)`, presumably for side-effect, then jumps to the join point, labelled `lj`. If, however, `x` is even, then control branches to the right child (labelled `lb`), doing no variable update. This vertex performs the Scheme call `(g x)` for side-effect, then branches to the join point, binding `y` to the square of `x` at the `lj` jump. Note that the join point is in the scope of loop variable `y`, since all paths to it define `y`.

Consider the control-dominator tree for the graph in Figure 6. The immediate dominator of vertices `la`, `lb` and `lj` is the initial `(odd? x)` vertex. Note that this dominator tree is expressed directly by the `let*` structure of the CFG term; this is ensured by the LC scoping used for code labels. If two vertices wish to jump to a common successor (such as the `lj` join point in our example), that successor must be bound to a label by a `let*` outside/above the two vertices. It is a general property of the CFG language that the binding structure of the labels provides a conservative approximation to the control-dominator tree—which means that algorithms that process CFG terms don't need to bother performing complex dominance-frontier calculations to determine the control-dominance relation. This is an intriguing contrast with other first-order, "flat" control representations, such as SSA.

5.4 Scope independence and control nondeterminance

The core CFG forms `let*`, `letrec`, `go` and `do` capture the idea of textually expressing a control-flow graph that can be composed in a structured way. The remaining `indep`, `permute` and `permute/tail` forms are included in the language to allow us to compose graph structure in ways that insulates the parts from one another. The `indep` form allows us to do parallel updates to loop variables. Each Scheme expression exp_i is evaluated in the loop-var scope of the `indep` form. The exp_i expression must produce as many return values as there are loop variables in the $vars_i$ binding list. After *all* the exp_i expressions have been evaluated, we bind the values to the variables in the $vars_i$ binding lists. The different binding lists must be disjoint. Thus no exp_i "sees" the variable updates made by any other exp_j.

The permute forms allow for non-deterministic control permutation (Figure 7). When control reaches a

$$\texttt{(permute ((}l_1\ cfg_1\texttt{) ...)}\ cfg\texttt{)}$$

form, the machine may arbitrarily permute the sequence of cfg_i terms. Then these terms are wired together in the permuted sequence; in cfg_i, a reference to label l_i is connected to the next clause in the sequence; the final clause has its l_i label connected to the body of the `permute` form, *cfg*.

Since variable scope in the CFG language is a function of control structure, the relaxed control spec for a `permute` form has a corresponding effect on its environment structure. Since any clause can come first in the dynamic execution order of the form, the updates performed by the other clauses do not contribute to the scope of cfg_i—but they *do* contribute to the scope of the final *cfg* clause. However, note that if the entire `permute` form is in the

```
(let* ((join  (permute ((l1  cfg1)
                        ...
                        (lk  cfgk))
               cfg))
       (left  (permute/tail ((l1  cfg1)
                             ...
                             (li  cfgi))
               (go join)))
       (right (permute/tail ((l1  cfg1)
                             ...
                             (lj  cfgj))
               (go join))))
  ...)
```

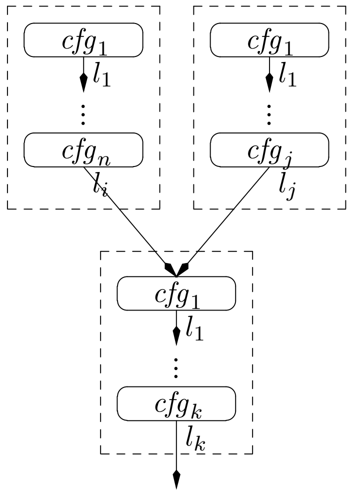


Figure 8. Tail-permute allows interpermuting across join points.

```
(permute ((l1  cfg1)
          ...
          (ln  cfgn))
  cfg)
```

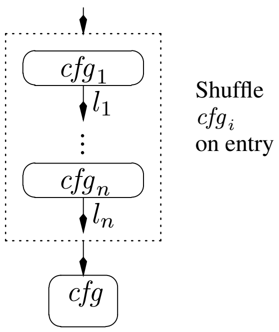


Figure 7. The `permute` form allows its constituent cfg_i terms to be executed in a non-deterministically permuted order before proceeding to the final cfg term. Each cfg_i term is threaded forwards to the following terms by means of the associated l_i exit label.

scope of loop variable `x`, then all the clauses are in the scope of `x`, and so an update to `x` performed by sub-graph cfg_i *will* be visible to all clauses that dynamically come after its execution. (This can be useful when two permutable updates to the same variable commute with one another, such as adding two elements to a common set.)

Finally, the `permute/tail` form allows for permutable sequences to extend across join points (Figure 8). The body cfg of a `permute/tail` form is restricted to be (1) a `permute` or `permute/tail` form, (2) a `(go` l`)` form that references a legal `permute/tail` body, or (3) a `let*` or `letrec` form whose body is a legal `permute/tail` body. When control reaches a `permute/tail` form, we chase down the chain of `permute/tail` forms that begin with this one until we reach the terminating `permute` form. The clauses of all of these forms are then permuted together to assemble the sequence to be executed.

The semantic specification of `permute` and `permute/tail` does not mean that an implementation is required to flip coins and shuffle blocks of control structure at run time. A typical implementation will freeze the sequence order of the relevant subterms at compile time. However, even if the order of execution is so frozen, the variable scoping retains its permutation-restricted semantics. The reasons for the permutable semantics are (1) isolating sub-terms from each other's scope contributions and (2) allowing the higher-level macros that produce the components to be given an order-independent control and scope semantics. We'll see how `permute` and `permute/tail` forms contribute to loop construction in following examples.

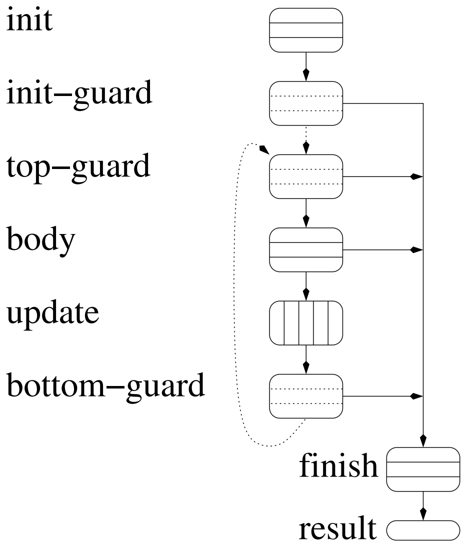
6. The LTK language and the loop template

Recall the point of the CFG language: it allows separate clauses in the loop form we are designing to provide pieces of control structure that can be plugged into a master template to assemble a complete loop. The general loop template is shown in Figure 9, along with the corresponding grammar for the "Loop Toolkit" (LTK) language we use to specify pieces of control-flow graph tagged with their destination in the template.

By design contract, the macros that produce component pieces of CFG structure to be inserted into the loop template specify control linkages by generating CFG terms that have up to three free labels: `p`, `s` and `f`. Jumping to a `p` label is used to "proceed" with normal execution of the loop; jumping to an `s` label is used to "skip" to the next iteration of the loop; jumping to an `f` label is used to "finish" or terminate the loop.

The semantics of the distinct components of the loop template are as follows:

- **Init**
  The init block contains the parts of the loop prologue that should execute unconditionally. Each such CFG form should have a single free label, `p`, used to string multiple `init` components together; jumps to `s` and `f` labels are not allowed.
- **init-guard**
  The init-guard block contains loop-initialisation code which may cause the loop to terminate with no iterations at all. An init-guard CFG term may have references to free labels `p` or `f` (but not `s`). The `f` references in the init-guard CFGs are connected to the finish block, while the `p` labels are used to continue executing the other init-guard terms before proceeding to the top-guard block.
- **Top-guard**
  The top-guard block contains fragments that perform the per-iteration termination tests that should occur at the beginning of each iteration. The individual cfg terms are allowed to refer to free `p` and `f` labels, with the `f` labels jumping to the finish block, and the `p` labels being used to string together the various top-guard cfgs.




```
ltk ::= (init cfg)
      | (init-guard cfg)
      | (top-guard cfg)
      | (body cfg)
      | (update vars exp)
      | (bottom-guard cfg)
      | (finish cfg)
      | (result exp)
```

Figure 9. The loop template and its corresponding LTK language. Dotted lines indicate permutable sequences.

- **Body**
  The body block contains the fragments that are intended to comprise the per-iteration body of the loop. Besides `p` and `f` labels, these fragments are also allowed to refer to `s` labels, which are connected to short-cut transfers to the update block, causing execution to skip the rest of the body block.
- **Update**
  The update block is for performing loop-variable updates. These independent update components are composed together using an `indep` CFG term, thus causing updates to be done in parallel.
- **Bottom-guard**
  The bottom-guard block is the location for bottom-of-the-loop per-iteration termination tests and conditional updates. Bottom-guard cfgs are connected into the template using their `p` and `f` labels.
- **Finish & result**
  The finish block is for wrap-up code, and the result block is for the Scheme expression that provides the final value of the loop to its containing Scheme expression. Finish cfgs may only refer to free `p` labels. There can be only one `result` form in a complete LTK program.

The three guard blocks—init-guard, top-guard and bottom-guard—are constructed using `permute` and `permute/tail` to assemble the block from its respective LTK components. This means that scope-introducing guard clauses are order-independent, which is important to allow loop clauses to have position-independent scoping properties.

To give an idea for how we can use the LTK language to define loop clauses, consider the loop form's `for` and `incr` clauses. The loop clause

```
(loop ...
      (for x in exp)
      ...)
```

causes variable `x` to sequence through the elements of the list exp. We implement this by having the `for` clause expand into the pair of LTK forms:

```
;;; TMP fresh / X from FOR clause.
(init (do (λ (e) (e exp))
          ((tmp) (go p))))
(top-guard (do (λ (e1 e2) (if (pair? tmp)
                              (e1 (car tmp)
                                  (cdr tmp))
                              (e2)))
               ((x tmp) (go p))
               (() (go f))))
```

where the loop variable `tmp` is a fresh variable generated using Scheme's hygienic macro system. This use of hygiene illustrates a general principle behind the definition of loop clauses. Private, intra-clause state, that must be communicated between, for example, the `init` and `top-guard` blocks of code produced by a single clause is managed by having the macro producing the LTK forms create a fresh variable (such as `tmp`) and insert it into both LTK forms. Because this variable is fresh, it cannot clash with or be referenced by code produced by any other loop clause. The same is true of the exit-edge Scheme variables bound by the *proc* subform in a `do` CFG term (`e`, `e1` and `e2` in the example)—these variables are typically fresh variables created by the macro that constructs the `do` term around code taken from a loop-clause term, which was written by the original author. Thus these exit-edge variable bindings can't accidentally capture variable references in the user code. In contrast, the loop variable `x` is *inter*-clause state—presumably the reason it is being defined in the loop's `for` clause is so that some other clause can refer to it. We manage this kind of state, linking the producer `for` clause with a consumer clause, by means of common reference to the same variable—in this case, `x`.

Notice that the `for` clause *conditionally* binds `x`. Since it only has a defined value if the source list is non-empty, its definition happens below a conditional split in the clause's CFG.

Similarly, consider a loop clause that steps a variable across an arithmetic sequence, such as

```
(loop ...
      (incr i from (* j j) to (- k 3))
      ...)
```

```
(: ini₁ ...                                                    ; init
   (let* ((f (: fin₁ ... (do (λ () res)))))                    ; finish & result
     (permute/tail ((p ig₁) ...)                               ; init-guard
       (letrec ((p (permute ((p tg₁) ...)                      ; top-guard
                      (let* ((s (indep ((upvars₁ upexp₁) ...) ; update
                                   (permute/tail ((p bg₁) ...) ; bottom-guard
                                     (go p)))))
                        (: body₁ ... (go s))))))               ; body
         (go p)))))
```

Figure 10. Putting it all together: inserting the LTK clauses into the master loop template to assemble a complete CFG. Comments on right side indicate the LTK clauses providing the CFG terms inserted on each line. The `(:` cfg_1 `...)` form is syntactic sugar for the p-serialising form `(let* ((p` cfg_n`) ... (p` cfg_2`))` cfg_1`)`. Note how label scope for each individual form is restricted to the allowed linkages for the LTK element.

The `incr` clause creates a fresh identifier `hi` and then expands into the trio of LTK forms

```
(init (do (λ (e) (e (* j j) (- k 3)))
          ((i hi) (go p))))
(top-guard (do (λ (e1 e2)
                 (if (< i hi) (e1) (e2)))
               (() (go p))
               (() (go f))))
(update (i) (+ i 1))
```

As with the `for` clause above, the `incr` clause's three LTK forms are stitched into the loop template by means of their references to free `p` and `f` labels.

7. Rotating loop tests and permutable loop blocks

We saw one possible definition of the `(for` *var* `in` *list-exp*`)` loop clause in the pair of `init` and `top-guard` LTK terms shown above. But there is another completely reasonable definition we could use, which rotates the conditional test back along the loop template from the top-guard block into the init-guard and bottom-guard blocks, replicating the code:

```
;;; (for var in list-exp)
(init         (do (λ (e) (e list-exp))
                  ((tmp) (go p))))
(init-guard   (do (λ (e1 e2) (if (pair? tmp)
                                 (e1 (car tmp)
                                     (cdr tmp))
                                 (e2)))
                  ((var tmp) (go p))
                  (()        (go f))))
(bottom-guard (do (λ (e1 e2) (if (pair? tmp)
                                 (e1 (car tmp)
                                     (cdr tmp))
                                 (e2)))
                  ((var tmp) (go p))
                  (()        (go f))))
```

We might wish to define `for` clauses this way to put the per-iteration test at the bottom of loop, thus allowing the compiler to save one branch instruction when generating assembler code for the loop by combining the termination test's conditional branch with the branch that jumps back to the top of the loop. This is a common compiler trick.

The problem is that we've changed the control-flow graph. Control structure, in our CFG model, determines environment structure. So our CFG change induces a change in scoping. In the rotated definition, *var*'s definition is pulled up to the init-guard block, which places top-guard code in the scope of *var*.

It's a design problem that these two implementations of the `for` clause provide different variable scope—we'd like the meaning of the loop to be invariant across the two definitions. We can restore semantic invariance by making the init-guard, top-guard and bottom-guard elements all interpermutable. We do this by inserting the init-guard and bottom-guard elements into `permute/tail` forms; these two blocks transfer control to the top-guard block, which is constructed as a `permute` form. That is, suppose the various clauses of a loop form produce n init-guard LTK terms `(init-guard` ig_1`)` ... `(init-guard` ig_n`)`. Then the init-guard block is

```
(permute/tail ((p ig₁)  ; permuted
               ...      ;  init-guard
               (p igₙ)) ;  elements
  loop-top) ; top-guard permutes
```

Note how the `p` labels are bound by the `permute/tail` form to capture the exit of each *ig* CFG term. In turn, if the various top-guard LTK forms produced by the loop clauses are `(top-guard` tg_1`)` ... `(top-guard` tg_n`)`, then the top-guard CFG block is rendered as

```
(permute ((p tg₁) ; top-guard block
          ...
          (p tgₙ))
  loop-body) ; body/update/bottom-guard blocks
```

Within the loop body, the bottom-guard elements are assembled into a `permute/tail` just as the init-guard terms are; the init-guard and bottom-guard `permute/tail` forms share the same target, the top-guard `permute` form.

This means that when control arrives at the top of the init-guard block, in principle, all the init-guard and top-guard terms are collected together and permuted; we then execute the permuted sequence of tests. So definitions introduced by these terms do not introduce scope visible by the other init-guard and top-guard clauses (although their definitions *do* introduce scope for the loop body itself). Similarly, when control reaches the bottom-guard block, all the bottom-guard and top-guard LTK terms are collected together and interpermuted. Thus it doesn't matter if a conditional test occurs in the shared top-guard `permute` block, or is replicated back into the two init-guard and top-guard `permute/tail` prefixes. A loop-clause implementor can choose either implementation with no change in the specified scope or control behavior.

If all this interpermuting seems a bit complex, bear in mind that this is simply the hidden low-level semantics providing the building blocks we use to construct the higher-level control and environment fragments—the loop clauses—that the programmer actually uses to construct a loop. These details ensure that these fragments "fit" together at the loop-clause level in a clean and modular way. In particular, allowing for control permutation provides order-independent scoping that simplifies the semantics at the loop-clause level.

The complete template for the loop form is shown in Figure 10; this is the graphical template of Figure 9 rendered as a CFG term.

$$E, [\![\texttt{(let ((}l_1\ \ cfg_1\texttt{))}\ \ cfg\texttt{)}]\!] \rightsquigarrow E, [l_1 \mapsto cfg_1]\, cfg$$

$$\begin{array}{l} E, [\![\texttt{(letrec ((}l_1\ \ cfg_1\texttt{)}\ \ldots\texttt{)}\ \ cfg\texttt{)}]\!] \rightsquigarrow \\ \qquad E, \left[l_i \mapsto [\![\texttt{(letrec ((}l_1\ \ cfg_1\texttt{)}\ldots\texttt{)}\ \ cfg_i\texttt{)}]\!]\right] cfg \end{array}$$

$$\frac{E, prim \xrightarrow{\text{prim}} (i, \langle v_1, \ldots, v_j \rangle)}{E, \left[\!\!\left[\begin{array}{l} \texttt{(do}\ prim\ \texttt{((}x_{1,1}\ \ldots\ x_{1,k_1}\texttt{)}\ \ cfg_1\texttt{)} \\ \qquad\quad \ldots \\ \qquad\quad \texttt{((}x_{n,1}\ \ldots\ x_{n,k_n}\texttt{)}\ \ cfg_n\texttt{))} \end{array}\right]\!\!\right] \rightsquigarrow E[x_{i,j} \mapsto v_j], cfg_i} \quad \begin{array}{l} 1 \le i \le n \\ j = k_i \end{array}$$

$$E, [\![\texttt{(permute ()}\ \ cfg\texttt{)}]\!] \rightsquigarrow E, cfg$$

$$\begin{array}{l} E, [\![\texttt{(permute (}b_1 \ldots b_n\texttt{)}\ \ cfg\texttt{)}]\!] \rightsquigarrow \\ \qquad E, [\![\texttt{(let ((}l_i\ \texttt{(permute (}b_1 \ldots b_{i-1}\ \ b_{i+1} \ldots b_n\texttt{)}\ \ cfg\texttt{)))}\ \ cfg_i\texttt{)}]\!] \\ \qquad \text{where } b_i = [\![\texttt{(}l_i\ \ cfg_i\texttt{)}]\!] \end{array}$$

$$\frac{E, cfg \rightsquigarrow E', cfg'}{\begin{array}{l} E, [\![\texttt{(permute/tail (}b_1\ \ldots\texttt{)}\ \ cfg\texttt{)}]\!] \rightsquigarrow \\ \qquad E, [\![\texttt{(permute/tail (}b_1\ \ldots\texttt{)}\ \ cfg'\texttt{)}]\!] \end{array}} \quad \begin{array}{l} cfg \text{ is } \texttt{let} \\ \text{or } \texttt{letrec}. \end{array}$$

$$E, \left[\!\!\left[\begin{array}{l} \texttt{(permute/tail (}b_1 \ldots\texttt{)} \\ \quad \texttt{(permute/tail (}b_1' \ldots\texttt{)} \\ \qquad cfg\texttt{))} \end{array}\right]\!\!\right] \rightsquigarrow E, [\![\texttt{(permute/tail (}b_1 \ldots b_1' \ldots\texttt{)}\ \ cfg\texttt{)}]\!]$$

$$E, \left[\!\!\left[\begin{array}{l} \texttt{(permute/tail (}b_1 \ldots\texttt{)} \\ \quad \texttt{(permute (}b_1' \ldots\texttt{)} \\ \qquad cfg\texttt{))} \end{array}\right]\!\!\right] \rightsquigarrow E, [\![\texttt{(permute (}b_1 \ldots b_1' \ldots\texttt{)}\ \ cfg\texttt{)}]\!]$$

Figure 11. The transition relation for the dynamic semantics of the CFG language.

8. Formal semantics

By this point, we've had enough informal description and seen enough examples to define a simple small-step operational semantics, giving a precise meaning to the CFG language. We can represent a machine configuration as a term in the language plus an environment E giving the current values of the defined variables. Note that the environment gives the values of *all* variables defined by prior execution, not just the ones in scope at the current control point—we are not yet defining any notion of "loop-variable scope." We don't need to define the language in terms of an actual graph, because joins and cycles in the graph structure are defined by means of labels, which are managed using the standard scoping mechanisms of the λ-calculus. Thus a simple substitution model suffices to "unroll" the term on demand as execution proceeds through the control-flow graph. Unrolling is managed by means of *label substitutions*: a substitution $[l_1 \mapsto cfg_1, \ldots, l_n \mapsto cfg_n]$ is a map from labels to CFG terms that is the identity function at all but a finite number of elements in its domain. Label substitution is lifted to CFG terms in the usual capture-avoiding way.

We also need a set of rules to model the primitive computations sited at the graph nodes (which is provided by the Scheme code in the `do` and `indep` forms of the concrete CFG language). In our formal semantics, we model these computations with a $\xrightarrow{\text{prim}}$ relation that relates an environment/primitive pair $E, prim$ to an edge-index/value-vector pair $(i, \langle v_1, \ldots, v_j \rangle)$. Such a relation means that if primitive computation $prim$ is performed starting with variable context E, it finishes by producing the vector of values $\langle v_1, \ldots, v_j \rangle$, and electing to proceed along exit edge #i. We only model `do` in this semantics; `indep` is a trivial variant.

With these pieces in place, we can define our transition relation $\rightsquigarrow$ with the schema shown in Figure 11. The relation is fairly simple. The first two rules handle `let` and `letrec` terms by completely standard substitution steps, with the latter unrolling the recursion once. The `do` rule simply fires the primitive computation, then traverses the indicated edge, while making the indicated loop-variable updates. The only rules of any real interest are the remaining rules, for `permute` and `permute/tail`. These assemble permutable items from `permute/tail` chains, and nondeterministically execute the elements.

The rules make it clear how completely the language is focussed on control and environment manipulation. All the real computation is left to the $\xrightarrow{\text{prim}}$ relation; the CFG terms provide the control and environment "glue" that connects the primitive computations. The rules also show how closely the CFG language is related to CPS; one indicator is their lack of recursion—note that the premise of the first `permute/tail` rule is not truly recursive, but simply a device for compactly writing down what would otherwise be a pair of more verbose axiom schema (for `permute/tail` terms with `let` and `letrec` bodies, respectively).

9. Types for scope

Something is missing from our dynamic semantics: scope. It appears nowhere in the schema for the $\rightsquigarrow$ relation. The semantics, as defined by these schema, simply runs each primitive computation in the context of whatever definitions happen to have dynamically occurred during the execution that led to that primitive's particular `do` form.

$$\frac{}{\Sigma, \Delta \vdash [\![\texttt{(go } l\texttt{)}]\!] : \sigma} \quad \begin{array}{l} \Sigma l = \langle \sigma', \delta' \rangle \\ \sigma' \subseteq \sigma \\ \Delta l \subseteq \delta' \end{array}$$

$$\frac{\Sigma', \Delta' \vdash \mathit{cfg} : \sigma \qquad \Sigma, \Delta'' \vdash \mathit{cfg}_1 : \sigma_1}{\Sigma, \Delta \vdash [\![\texttt{(let ((}l_1\ \delta_1\ \sigma_1\ \mathit{cfg}_1\texttt{)) } \mathit{cfg}\texttt{)}]\!] : \sigma} \quad \begin{array}{l} \Sigma' = \Sigma[l_1 \mapsto \langle \delta_1, \sigma_1 \rangle] \\ \Delta' = \Delta[l_1 \mapsto \emptyset] \\ \Delta'' = \Delta \cup \lambda\_.\delta_1 \\ \delta_1 \subseteq \sigma_1 \end{array}$$

$$\frac{\Sigma, \Delta_1 \vdash \mathit{cfg}_1 : \sigma_1, \ \ldots}{\Sigma, \Delta \vdash [\![\texttt{(do } \sigma'\ \mathit{proc}\ \texttt{(}\mathit{vars}_1\ \mathit{cfg}_1\texttt{)} \ \ldots\texttt{)}]\!] : \sigma} \quad \begin{array}{l} \Delta_i = \Delta \cup \lambda\_.\mathit{vars}_i \\ \sigma_i = \sigma' \cup \mathit{vars}_i \\ \sigma' \subset \sigma \end{array}$$

$$\frac{\Sigma', \Delta' \vdash \mathit{cfg} : \sigma \qquad \Sigma', \Delta'' \vdash \mathit{cfg}_1 : \sigma_1}{\Sigma, \Delta \vdash [\![\texttt{(letrec ((}l_1\ \delta_1\ \sigma_1\ \mathit{cfg}_1\texttt{)) } \mathit{cfg}\texttt{)}]\!] : \sigma} \quad \begin{array}{l} \Sigma' = \Sigma[l_1 \mapsto \langle \delta_1, \sigma_1 \rangle] \\ \Delta' = \Delta[l_1 \mapsto \emptyset] \\ \Delta'' = \Delta \cup \lambda\_.\delta_1 \\ \delta_1 \subseteq \sigma_1 \end{array}$$

Figure 13. Scope-establishing type-judgement schema for core of type-annotated CFG language

$$\begin{array}{rcl} \mathit{tcfg} & ::= & \texttt{(let* ((}l_1\ \sigma_1\ \delta_1\ \mathit{tcfg}_1\texttt{) ...) } \mathit{tcfg}\texttt{)} \\ & \mid & \texttt{(letrec ((}l_1\ \sigma_1\ \delta_1\ \mathit{tcfg}_1\texttt{) ...) } \mathit{tcfg}\texttt{)} \\ & \mid & \texttt{(go } l\texttt{)} \\ & \mid & \texttt{(do } \sigma\ \mathit{proc}\ \texttt{(}\mathit{vars}_1\ \mathit{tcfg}_1\texttt{) ...)} \\ \sigma\texttt{,}\ \delta & ::= & \texttt{(}\mathit{ident}\ \texttt{...)} \end{array}$$

Figure 12. Syntax of core type-annotated CFG language.

Recall our fundamental scoping principle: scope proceeds from control. The semantics we've defined provides a control story; thus it implies one for scope, as well. The scope of a given `do` subterm in a CFG is simply the set of loop variables that are defined on every execution path to that subterm. With this definition, suitably formalised, we can then restrict the premise of the `do` rule so that a $\xrightarrow{\text{prim}}$ computation is only provided the statically-guaranteed subset of the dynamic variable environment at that point. That is, we can run the primitive computation in its proper scope.

Once a complete CFG has been constructed, the task of translating it into Scheme is mediated by an analysis step that determines the scope of each program point. The results of this analysis are used to annotate the CFG; these annotations guide the subsequent compilation step into final Scheme code. We can think of this step as a type-inference step, where our types express the environment structure of CFG terms. However, the calculation does not have the structure of standard control-dominance algorithms, because we can use the lexically explicit label-scoping structure to conservatively approximate the dominance tree.

To simplify the presentation, we first present the type system for the core CFG language, without the `permute`, `permute/tail` and `indep` forms. The grammar of the type-annotated language is shown in Figure 12. Essentially, we tag each label and `do` form in the language with σ and δ identifier sets:

- σ**:** Vars *definitely* defined (all paths) at this control point.
- δ**:** Vars that *might be* defined (some path) between label l's binder (*i.e.*, l's immediate dominator) and the labelled control point.

The basic type judgement depends on two type environments:

$$\begin{array}{rcl} \Sigma & : & \mathit{Lab} \to \mathit{Scope} \times \mathit{Defs} \\ \Delta & : & \mathit{Lab} \to \mathit{Defs} \\ \mathit{Scope} & = & \mathcal{P}(\mathit{Ident}) \\ \mathit{Defs} & = & \mathcal{P}(\mathit{Ident}) \end{array}$$

Σ maps a label to the type information to which it is statically bound. For a given control point, Δ maps a free label to the set of variables that may have been defined on *some* path between that label's binding point and the given control point.

The basic type judgement establishing the scope of an annotated term is

$$\Sigma, \Delta \vdash \mathit{tcfg} : \sigma.$$

The type judgement is defined with the schemas of Figure 13; the `let*` and `letrec` rules are simplified to handle a single bound label, but the generalisation to the full form is straightforward.

We extend the type system to handle permutable sequences by extending the σ scope types to permit a *pair* of loop-variable sets ($\sigma_{pre}\ \sigma_{perm}$) to specify the type of a `permute` or `permute/tail` context. When a term has such a type, it means that the term is an element of a `permute/tail` chain. The σ_{pre} set specifies the scope that was visible at the start of the chain (and, hence, at each sub-term of all `permute` and `permute/tail` forms in the chain), while the σ_{perm} set specifies the extra scope being accumulated by preceding elements of the chain to be eventually made visible to the term at the end of the chain.

Examining the rules for the type system, it's not hard to see that it captures the notion of "variables that are defined on all paths to a term." To express this formally, we first extend the type judgement to machine configurations, where the semantics is altered to operate upon typed terms. We say that a machine configuration is valid, written $\vdash E, \mathit{tcfg}$, if $[], [] \vdash \mathit{tcfg} : \mathrm{dom}(E)$. That is, a machine configuration is valid if its type—that is, its scope—is satisfied by the current dynamic environment. The key theorem, then, is a kind of preservation theorem:

THEOREM 1. $\vdash E, \mathit{tcfg} \ \wedge\ E, \mathit{tcfg} \leadsto E', \mathit{tcfg}' \ \Rightarrow\ \vdash E', \mathit{tcfg}'$.

The proof of the theorem is by induction on the justification tree for the typing of the tcfg term, in the standard form for type-preservation proofs. It relies on another straightforward lemma, that label substitutions preserve type; again, this can be shown by induction.

From our theorem, and the typing rule for `do` forms, it follows that if we execute a well-typed term in an initial empty environment, that whenever execution reaches a `do` form, every variable in the form's scope will have a definition in the current environment. Thus we have a solid definition of static variable scope in terms of our original BDR principle.

While beyond the scope of this paper, we can perform a similar formalisation to capture the meaning of the δ declarations.

10. Type-inference and type-directed translation

We infer the type annotations for a complete CFG term by assigning a type variable to each position in the syntax where a σ or δ should be, then recursing once over the tree to collect set-inclusion constraints generated by the type schema, then propagating information around the constraints to find maximal types.

Once we have performed the type inference, it is fairly straightforward to translate the annotated term to Scheme code. The compiler (a Scheme macro) is guided by the σ and δ annotations. A labelled CFG point is translated to a Scheme variable let-bound to a procedure, using the following conventions:

- We assume the procedure's body is closed in a Scheme scope that sees all loop variables in the term's scope.
- The procedure has for its parameters all variables that might have been defined/updated between the time the procedure was evaluated and the time it is applied. This is exactly the δ set for the label.
- A `(go` l`)` form is translated to a call to l's Scheme procedure, passing the current values of l's δ set as parameters.

To prevent variable capture, the Scheme variables used to name control points in the generated code are fresh names.

Permutable sequences are compiled by means of maintaining a parallel set of fresh Scheme variables for the loop variables, called "shadow" variables. If variable `x` is in the σ_{perm} scope of a permute chain, then a definition of `x` that happens during execution of a component of the chain is bound to `x`'s shadow variable, so that subsequent elements of the chain do not see this binding. At the end of the chain, all the newly introduced scope is exposed by binding the newly introduced variables to their shadow-variable values.

The translation from a typed CFG to pure-functional Scheme code is really just a kind of SSA conversion [9]. However, it's much simpler than standard SSA-conversion algorithms—a few dozen lines of code suffice. One reason the translation is so simple is because, as we noted earlier, there is no need to compute dominance trees. The dominance information directly encoded in the syntactic structure and the type annotations provide all the information needed for the translation. These properties of the CFG language make it an interesting possibility for a compiler intermediate representation in other settings.

11. Implementation

The current implementation is a fairly ambitious Scheme macro, implemented using a mix of high-level R5RS macros [10] and low-level Clinger-Rees "explicit renaming" macros [3]. It is comprised of about 4500 lines of highly-commented code (about 2800 lines excluding blank lines and comments). This breaks down, very roughly, as follows:

1000	definitions of individual loop clauses
350	loop→ltk macro core
115	ltk→cfg macro
280	CFG AST, sexp/AST parsing & unparsing
320	CFG simplifier
890	type inference
420	tcfg→scheme compiler[2]
500	general utilities

Somewhat unusually for a Scheme macro, the CFG-processing code does not operate on raw s-expressions. Instead, it parses the incoming s-expression into an AST comprised of records, then operates on the AST. This is just reasonable software engineering: compilers of a certain complexity need real data structures for their intermediate representations. The entire system is strongly modularised by using the various languages as "hinge points." As most of the system is concerned with translating expressions in one non-Scheme language to another non-Scheme language (*e.g.*, between loop clauses and LTK forms), the system makes tremendous use of "CPS macros" [8], a style of macro useage which permits a Scheme macro to target a non-Scheme language. Of the 83 macros defined in the source, 75 are CPS macros.

I expect the line count to increase 50-100% before the package is released, for two reasons. First, the system works well when the programmer makes no errors. But static semantics and especially syntax errors produce incomprehensible error messages. This is due to the fact that little syntax checking is actually *programmed* into the system; it mostly comes from the pattern-matching machinery of the various CPS macros that process the language. Worse, some errors won't manifest themselves until multiple transformations have happened to the original form. This can be handled, again, by reasonable software engineering common to any compiler: error checking must be performed as early as possible, and as high in the language tower as possible. It is frequently the case with robust, industrial-strength software systems for error-handling code to dominate the line counts; the loop package is no different. Adding the code to provide careful syntax checking and clear error messages is tedious but straightforward implementation work; it will be required to turn the initial version into a tool that is generally useable.

Second, I have designed but not implemented a facility for describing general accumulators. These are provided by means of BDR-scoped macros, allowing loop-clause writers to implement "object-oriented" macros that obey a standard object protocol for accumulators. The state of each accumulator is private to the defining loop clause, but can be tightly integrated with the rest of the CFG loop state all the same, providing abstraction and modularity without sacrificing performance.

12. Quicksort example

We've focussed on the CFG language in this paper, primarily because the underlying framework provided by the CFG language is the chief intellectual contribution of the design. However, before finishing, we should present at least one real example of the top-level loop language in use. Here is in-place vector quicksort, written in Scheme with the loop package:

```
(let recur ((l 0) (r (vector-length v)))
  (if (> (- r l) 1)
      (loop (initial (p (pick-pivot l r))
                     (i (- l 1))
                     (j r))

            (subloop (incr i from i)
                     (bind (vi (vector-ref v i)))
                     (while (< vi p)))
            (subloop (decr j from j)
                     (bind (vj (vector-ref v j)))
                     (while (> vj p)))
            (until (<= j i))
            (do (vector-set! v i vj)
                (vector-set! v j vi))

            (after (recur l        i)
                   (recur (+ j 1)  r)))))
```

[2] I actually wrote three such compilers, of varying properties. The most complex version is 420 lines; the simplest, 136. Again, about half these lines counts are comments or blank lines.

This example manages to pack a fair amount of control structure into a few lines of code, containing one recursive function and three distinct loops. The `loop` form begins by initially binding three variables: the pivot value `p`, and the left and right indices of the partition step, `i` and `j`. (The pivot selection is performed off-stage by a `pick-pivot` function.)

The body of the loop is comprised of four clauses: two `subloop` clauses, a `do` and an `until` clause. A `subloop` clause is used to perform a nested loop that shares loop variables with its containing loop (this is easy to arrange, given that our CFG notation handles general graph structure with no problem). The first `subloop` clause steps `i` from left to right, skipping over indices whose element is less than the pivot value. On each iteration of the subloop, we increment `i`, then bind `vi` to the element of vector `v` at that index, then compare `vi` and the pivot `p`. (A fine point: the `from` keyword causes an `incr` or `decr` clause to skip the first value; the more common case of including the initial value is provided by the alternate keyword `:from`. A similar variation with the `to` and `to:` keywords that provide the final value allows a programmer to specify closed, open, or half-open intervals. Thus the simple mnemonic: including the colon includes the end point; leaving it off, omits the end point.) If `vi` $\geq$ `p`, the subloop is terminated. (Note one of the features of the loop design: a given loop can have multiple `while`/`until` termination tests; each test is performed where it occurs in the loop, interleaved with other body clauses.) Thus, the subloop terminates with `i` bound to the leftmost index of `v` whose element is greater than or equal to `p`. Similarly, the second subloop steps `j` from right to left, terminating when `j` reaches an element of `v` that is less than or equal to `p`. The next clause in the main loop body, the `until` clause, checks to see if the partition indices have overlapped; if so, the partition loop is done. If not, the loop continues on to the `do` clause. A `do` clause is a body element whose embedded Scheme code is simply executed for side effect. This particular one swaps the two elements at the `i` and `j` indices, after which the loop proceeds to the next iteration. (Note that the swap code executes in the scope of the `vi` and `vj` bindings, since they occur before the termination tests of their respective subloops.) When the partition loop is done, its `after` clause recurs on the two segments of the partition.

13. Related work

Iteration is one of the fundamental things programmers do; as a result, there is a wealth of related work on defining language forms for specifying loops. Haskell's list comprehensions [19] are one alternative; Egner's SRFI-42 [6] provides some of its ideas in a call-by-value setting as "eager comprehensions." However, SRFI-42 is a more limited design than the loop form presented here. It does not permit the construction of arbitrarily complex control structure for its iterations. It's not clear, for example, how one could use the system to construct more than one result (*e.g.*, a list and some derived summary integer). On the other hand, limited notations can frequently be clearer in their restricted domain of applicability.

One of the charms of working with systems based on macros is their ability to incorporate further elements of specialised syntax. For example, suppose we wanted to add a new `in-table` keyword to the `for` clause allowing database queries, *e.g.*:

```
(loop ...
      (for row-vars in-table sql-query)
      ...)
```

We'd like to be able to use some s-expression form of SQL in the query part of the driver clause, rather than encoding the query as a string. One benefit is the increased static checking we get for our uses of the specialised notation [4]; to quote a Perlis aphorism [11], "The string is a stark data structure and everywhere it is passed there is much duplication of process. It is a perfect vehicle for hiding information." Macros are just little compilers, which makes them a powerful tool for programmers, allowing them to construct software systems by the composition of little languages [13]. To compare, we could not, for example, embed SQL notation into a Haskell comprehension. Egner's comprehension system, on the other hand, does have this property of linguistic composability, because it is also built with macros in an extensible manner.

Water's LetS (later renamed OSS, later renamed simply "Series") system [20, 21, 22, 18, 23] lets one write loops in an explicitly data-parallel style, rather like APL. The package attempts to process the computation in an on-line manner, without allocating intermediate buffer structures. However, the reasoning required to ensure on-line processing can be tricky and subtle. This critical element of the design, in fact, went through a somewhat tortured evolution in the history of the design, shifting over time from an initial design that was carefully restricted to guarantee on-line processing to a more general one that inserts temporary buffers into the produced code where the macro is unable to resolve the dependencies. Another issue with LetS/OSS iterations is that they don't nest.

APL [7] itself is an interesting model: it essentially replaces "time-like" loops with "space-like" aggregate operations on multi-dimensional arrays. This is a beautiful model; the cost of its elegance and simplicity, however, is the language's complete abandonment of any attempt to guarantee that computations can be performed on-line, without allocating potentially enormous intermediate collections.

There have been many designs for loop packages in the general style of the one presented here. Besides Common Lisp's official loop facility [18], there is also the `iterate` package [1, 2], which is similar, if somewhat cleaner. The top-level loop design presented here traces back to the "Yale loop," designed and implemented for Maclisp at Yale in the early 1980's. The Yale loop and the MIT `iterate` form are quite similar. These Lisp packages typically are not pure-functional, that is, they expand into code that side-effects variables. This renders them unsuitable in language contexts that do not permit variable assignment (*i.e.*, most modern functional languages), and is a significant barrier to optimisation even for those languages that do. One of the contributions of the framework provided by the CFG language is that it provides a variable-update mechanism for iteration state that nonetheless still permits a purely functional connection to the language with which it is mutually embedded. This is a subtle design point, worth noting. Classic Lisp loop packages all share the same issues with respect to variable scope discussed earlier, as well.

14. Conclusion

The loop form presented here is an exercise in design that comes from following a single foundational idea: describing iteration structures using control-flow graphs, and letting the control-flow dictate the scoping. The other novel elements of the system—the compositional notation for encoding the control-flow graphs, the concept of permutable sequences, and the use of a type system to express scope—all flow from this core idea. By hewing consistently to this concept, we obtain a system that enjoys an underlying conceptual integrity, and this is what enables the framework to be extensible.

While the implementation is sizeable, programmers do not have to write multi-thousand line compilers themselves to take advantage of the extensibility provided by the macro-based framework. A programmer who is implementing, for example, a hash-table package, can add an `in-hash-table` keyword for the loop `for` clause to his module with about ten lines of code; clients of the package can then easily iterate over hash tables with loop forms. By building on the foundation provided by the CFG language, loop-clause designers have a simple notational "interface" to the semantic structures of control and scope that the notation encodes, as well as the (multi-thousand line) reasoning engines that process the notation. This is the power of the "language towers" approach to extensibility.

Note, also, that programmers typically do not need to be aware of the low-level scoping details of the CFG terms to which their loop forms translate. This is a crucial design criterion. It's not enough simply to have a scoping principle that is well defined in some formal sense. If a programmer were required to mentally construct the full control-flow graph for a given loop and then solve the associated dominance equations in his head, just to resolve a scope issue, the system would be *well defined*, but it would not be a *useful* tool for humans.

Instead, loop programmers *do* need to be aware of the general eight-block loop skeleton, because the scoping of various parts of loop clauses can be described in terms of this structure. For example, a variable first defined in the top-guard block of a loop is visible to the loop's body block, but not its final block. In short, the large-grain control structure of the loop skeleton constrains the scoping of the loop clauses to simple structures. Again (to stress the design message of this paper), this is no accident: the CFG semantics was carefully designed to allow loop-clause designers to provide scope and control semantics for their clauses in terms of this simple, large-scale structure.

Acknowledgements

In August 2004, Olivier Danvy convened a group of researchers at the University of Åarhus for an informal workshop on aspects of continuations and continuation-passing style representations. This work was stimulated and refined by discussion with the other participants: Andrzej Filinski, Mayer Goldberg, Julia Lawall and Olivier. In Boston, similar discussions with Alan Bawden, Matthias Felleisen, Greg Morrisett and Mitch Wand over extended periods of time helped push this work forwards. I am also grateful to Olivier for the invitation to spend the Fall of 2004 visiting at the University of Århus, which provided me the perfect balance of quiet reflection and collegial interaction needed to bring these ideas to completion.

The CFG design was first presented as a talk at *Daniel P. Friedman: A Celebration* in December 2004; during the years that I've been thinking about the definition of loop forms, I was guided by a remark Dan once made to me about the design of language forms: "Scope is everything." The entire structure of the CFG language proceeds from the focus provided by that single remark.

The top-level design of the loop form presented in this paper was strongly influenced by the Yale loop, designed for Maclisp by the Tools Group graduate students and faculty in Yale's Computer Science Department in the early 1980's; among others, John Ellis and Chris Riesbeck were primary contributors to that design. I have drawn on a long email message that Ellis wrote during that time to make the case for loop forms and against low-level tail recursion in Section 1.

References

[1] Jonathan Amsterdam. The Iterate manual. AI Memo 1236, MIT AI Lab, October 1990.

[2] Jonathan Amsterdam. Don't Loop, Iterate. Working Paper 324, MIT AI Lab.

[3] William Clinger and Jonathan Rees. Macros that work. In *Proceedings of the ACM Conference on Principles of Programming Languages,* pp. 155–162, 1991.

[4] Ryan Culpepper, Scott Owens and Matthew Flatt. Syntactic abstraction in component interfaces. In *Proceedings of the Fourth International Conference on Generative Programming and Component Engineering* (GPCE'05), September 2005, Tallinn, Estonia.

[5] Edsger W. Dijkstra. GOTO statement considered harmful. Letter to the Editor. *Comm. ACM* 11(3), March 1968.

[6] Sebastian Egner. Eager comprehensions in Scheme: The design of SRFI-42. In *Proceedings of the ACM SIGPLAN 2005 Workshop on Scheme and Functional Programming* (Scheme 2005), September 2005, Tallinn, Estonia. See also `http://srfi.schemers.org/srfi-42/`.

[7] Leonard Gilman and Allen J. Rose. *Apl: An Interactive Approach.* John Wiley & Sons Inc., Third edition, January 1984.

[8] Erik Hilsdale and Daniel P. Friedman. Writing macros in continuation-passing style. In *Proceedings of the Workshop on Scheme and Functional Programming* (Scheme 2000), pages 53–61, Montreal, Canada September 2000. Rice Technical Report 00-368. September 2000.

[9] Richard A. Kelsey. A correspondence between continuation-passing style and static single assignment form. In *Proceedings of the ACM SIGPLAN Workshop on Intermediate Representations*, *SIGPLAN Notices* 30(3), pages 13–22, 1995.

[10] Richard Kelsey, William Clinger and Jonathan Rees (Editors). Revised^5 report on the algorithmic language Scheme. *SIGPLAN Notices*, 1998.

[11] Alan J. Perlis. Epigrams on programming. *Sigplan* 17 #9, September 1980.

[12] Kent M. Pitman. Common Lisp Issue LOOP-INITFORM-ENVIRONMENT:PARTIAL-INTERLEAVING-VAGUE. March 1991. `http://www.lisp.org/HyperSpec/Issues/iss222-writeup.html`

[13] A universal scripting framework. Olin Shivers. In *Concurrency and Parallelism, Programming, Networking, and Security,* Lecture Notes in Computer Science #1179, pages 254–265, Editors Joxan Jaffar and Roland H. C. Yap, 1996, Springer.

[14] Guy L. Steele Jr. and Gerald Jay Sussman. LAMBDA: The ultimate imperative. AI Memo 353, MIT AI Lab, March 1976.

[15] Guy L. Steele Jr. LAMBDA: The ultimate declarative. AI Memo 379, MIT AI Lab, November 1976.

[16] Guy L. Steele Jr.. Debunking the "expensive procedure call" myth. AI Memo 443, MIT AI Lab, October 1977.

[17] Guy L. Steele Jr. *RABBIT: A Compiler for SCHEME.* Masters Thesis, MIT AI Lab, May 1978. Technical Report 474.

[18] Guy L. Steele Jr. *Common Lisp: The Language.* Digital Press, Maynard, Mass., second edition 1990.

[19] Philip Wadler. List comprehensions (Chapter 7). In *The Implementation of Functional Programming Languages*, Editor Simon L. Peyton Jones, Prentice Hall, 1987.

[20] Richard C. Waters. LetS: An expressional loop notation. AI Memo 680A, MIT AI Lab, February 1983.

[21] Richard C. Waters. Obviously synchronizable series expressions: Part I: User's manual for the OSS macro package. AI Memo 958A, MIT AI Lab, March 1988.

[22] Richard C. Waters. Obviously synchronizable series expressions: Part II: Overview of the theory and implementation. AI Memo 959A, MIT AI Lab, March 1988.

[23] Richard C. Waters. Automatic transformation of series expressions into loops. *ACM Transactions on Programming Languages and Systems*, 13(1):52–98, January 1991. See also AI Memos 1082 and 1083, MIT AI Lab, 1989.