Monadic Intermediate Language Documentation

Mark P Jones

May 14, 2013

Contents

1 A Monadic Intermediate Language, MIL **4**

1.1 Abstract Syntax 4

1.1.1 Programs 4

1.1.2 Definitions 5

1.1.3 Monadic Code Sequences 8

1.1.4 Generalized Tail Calls 12

1.1.5 Primitives 16

1.1.6 Atoms 18

1.2 Dependency Analysis 21

1.2.1 Display Results in Graphviz Format 24

1.3 Display MIL Programs 24

1.4 Lists of Variables 27

1.5 Variable to Atom Substitutions 29

1.5.1 Representing and Constructing Substitutions 29

1.5.2 Applying Substitutions to Atoms 30

1.5.3 Applying Substitutions to Tail Values 30

1.5.4 Applying Substitutions to Code Values 31

1.6 An interpreter for MIL programs 32

1.6.1 Representing MIL values 32

1.6.2 Value Environments 33

1.6.3 Evaluator Methods 34

2 Optimization **36**

2.1 General Framework 38
2.2 Constructor Function Simplification 40
2.3 Derived Blocks 44
2.3.1 Adding a Trailing Invoke 47
2.3.2 Adding a Trailing Enter 50
2.3.3 Adding a Continuation 52
2.3.4 Specializing a Block on Known Constructors 56
2.4 Inlining 60
2.4.1 Return Analysis 61
2.4.2 Pre-Inlining Code Cleanup 63
2.4.3 Detecting Loops 64
2.4.4 Inlining Within Code Sequences 65
2.4.5 Prefix Inlining 67
2.4.6 Suffix Inlining 69
2.4.7 Inlining Tails 70
2.4.8 Rewrite BlockCalls to Skip Goto Blocks 71
2.5 Lifting Allocators 72
2.6 Eliminating Unused Arguments 75
2.6.1 Detecting Unused Arguments 77
2.6.2 Removing Unused Arguments 80
2.7 Simple Flow Analysis and Optimization 82
2.7.1 Facts 83
2.7.2 Main Flow Analysis 85
2.7.3 Liveness 89
2.7.4 Tracking Allocators at a BlockCall 90
2.7.5 Entering Function Closures Directly 90
2.7.6 Entering Monadic Thunks Directly 91
2.7.7 Shorting Out Matches 92
2.8 Optimizations for Primitives 93
2.8.1 Rewriting for Unary Primitives 94
2.8.2 Rewriting for Commutative Binary Primitives 95

2.8.3 Rewriting for Noncommutative Binary Primitives 100
2.8.4 Constant folding for relational operators 102
2.9 Eliminating Duplicate Blocks 104
2.9.1 Summarizing Code Sequences 104
2.9.2 Testing for Equivalent Sequences of Code 105
2.9.3 Identifying Duplicated Definitions 108
2.9.4 Rewriting 110
2.10 Analyzing Number of Calls 110

A Library Operations **113**

A.1 Singleton Classes 113
A.2 Caching 113
A.3 Simple List Operations 114
A.4 Ordered Lists 115
A.5 Dependency Analysis 118
A.5.1 Representing the Dependency Graph 119
A.5.2 The Forward Depth-First Search 119
A.5.3 Representing Binding SCCs 120
A.5.4 The Reverse Depth-First Search 122
A.5.5 Putting it All Together 123
A.5.6 Tracking Dependencies 124

Chapter 1

A Monadic Intermediate Language, MIL

MIL is a "monadic intermediate language" that is designed for use in the construction of optimizing compilers for both imperative and functional programming languages. This chapter provides a quick summary of MIL, describes a set of Java classes that are used to represent the constructs of its abstract syntax; and introduces some helper functions for working with MIL programs.

All of the code described in this chapter is part of the following Java package:

```
package mil;
```

1.1 Abstract Syntax

1.1.1 Programs

Every MIL program is described by a sequence of definitions:

```
p  ::=  def1; ...; defn
```

In the following, we will represent programs by objects of type `MILProgram`, each of which stores a list of entry points to the program. (By comparison, a standard C program typically has a unique entry point called `main`, whereas a library that is intended to be linked with and called from other code can be expected to have multiple entry points, one for each value that might be referenced from client code outside the library.) Any other definitions that are reachable from one or more of the entry points will still be considered as part of the program, even though they are not directly recorded in `MILProgram` objects. On the other

hand, definitions that are not used, and hence that are not reachable from any of the entry points, will be automatically dropped from the program as dead code.

```
public class MILProgram {
    /** Stores a list of the entry points for this program.
     */
    private Defns entries = null;

    /** Add an entry point for this program, if it is not already included.
     */
    public void addEntry(Defn defn) {
        if (!Defns.isIn(defn, entries)) {
            entries = new Defns(defn, entries);
        }
    }
}
```

1.1.2 Definitions

There are three different forms of definition that can be used in MIL programs, as shown by the following grammar:

```
def  ::=  b(v1, ..., vn)   = c    -- block entry point
       |  k{v1, ...; vn} v = c    -- closure entry
       |  v <- t                  -- initialize global
```

A definition of the form `b(v1, ..., vn) = c` corresponds to a basic block that starts at label `b`; expects `n` formal parameters to be passed in using the "registers" `v1`, ..., `vn`; and executes the sequence of instructions in the code sequence `c`. For example, the following block computes the sum of the squares of its two arguments (details about the code sequence on the right of the `=` sign will be described in a later section):

```
b(x,y) = u <- mul((x,x))      -- compute x*x in u
         v <- mul((y,y))      -- compute y*y in v
         w <- add((u,v))      -- add the two squares
         return w              -- return final result
```

Definitions of the form `k{v1, ...; vn} v = c` are used to specify the code sequence `c` that will be executed when we enter a closure with code pointer `k`; stored free variables `v1`, ..., `vn`; and an argument `v`. (We will shortly that this syntax mirrors the notation `k{v1, ...; vn}` that is used to describe the allocation of a corresponding closure value.) For example, given the following definition, we can use a closure with code pointer `k` and a stored free variable `n` to represent the function `(\x -> n + x)` that will return the value `n+x` whenever it is called with an argument `x`:

```
k{n} x = u <- add((n,x))
         return u
```

Finally, a definition of the form `v <- t` represents a binding for a top-level (i.e., global value) called `v` that is initialized using the result that is produced by executing the tail expression `t`. Note that values defined in this way are not the same as *global variables* in an imperative programming language because there is no way to change or otherwise assign a new value for an identifier `v` that has been introduced in this way.

These three different forms of definition are represented by values of the `Block`, `ClosureDefn`, and `TopLevel` classes, each of which is a subclass of an abstract base class called `Defn`. Note that names for individual blocks are generated automatically as strings of the form `bN` where a different integer `N` is used for each distinct block. In a similar way, closure definitions are assigned names of the form `kN`.

```
public abstract class Defn {

  // A block: b(formals) = code ----------------------------------
  public case Block(private Code code) {
      private static int   count   = 0;
      private final String id      = "b" + count++;
      private Var[]        formals; // filled in after construction by analysis
      setter formals;
  }

  // Closure definition: k{stored} arg = tail --------------------
  public case ClosureDefn(private Var arg, private Tail tail) {
      private static int   count  = 0;
      private final String id     = "k" + count++;
      private Var[]        stored /* = Var.noVars*/; // filled in after construction by analysis
      setter stored;
  }

  // Top-level defn: id <- code ----------------------------------
  public case TopLevel(private String id) {
      private Tail tail;

      /** Set the value associated with a top-level definition.
       */
      void setTopLevel(Tail tail) {
          if (this.tail!=null) {
              debug.Internal.error("Attempt to set a second top-level value for a TopLevel variable");
          }
          this.tail = tail;
      }

      // A constructor function: c <- function ----------------------
      // The code for a Cfun implements the actual constructor function.
      public case Cfun(private int arity, private int num, private Cfun[] constrs) {
          public getter arity, num, constrs;

          Cfun(String id, int arity, int num, Cfun[] constrs) > {
              constrs[num] = this; // Save this constructor in its array of constructors
          }

          public static final Cfun[] boolType  = new Cfun[2];
          public static final Cfun True        = new Cfun("True",    0, 1, boolType);
          public static final Cfun False       = new Cfun("False",   0, 0, boolType);

          public static final Cfun[] unitType  = new Cfun[1];
          public static final Cfun Unit        = new Cfun("Unit",    0, 0, unitType);

          public static final Cfun[] listType  = new Cfun[2];
```

```
        public static final Cfun Nil         = new Cfun("Nil",     0, 0, listType);
        public static final Cfun Cons        = new Cfun("Cons",    2, 1, listType);

        public static final Cfun[] pairType  = new Cfun[1];
        public static final Cfun Pair        = new Cfun("Pair",    2, 0, pairType);

        public static final Cfun[] maybeType = new Cfun[2];
        public static final Cfun Nothing     = new Cfun("Nothing", 0, 0, maybeType);
        public static final Cfun Just        = new Cfun("Just",    1, 1, maybeType);
    }
  }
}
```

Note that we also include "constructor functions", represented by the class `Cfun`, as a special form of `TopLevel` definitions. We will see shortly, for example, that expressions of the form `True()` and `False()` can be used to construct representations for the corresponding Boolean values, while expressions of the form `Pair(x,y)` are used to build a heap-allocated data structure with two subcomponents, represented here by `x` and `y`. In each case, the `arity` value describes the number of subcomponents/arguments that the constructor expects. It is also common to have multiple ways of constructing values of a single type. For example, there are two constructors for Booleans (`True` and `False`), but also two constructructors for lists (`Nil` and `Cons`, corresponding to empty and non-empty lists, respectively). We capture some of these details about individual constructors using the `num` and `constrs` fields of each `Cfun`: we assign a distinct number, `num`, for each constructor of a given type, and we record the set of distinct constructors for the underlying type in the 'constrs' field (with the expectation that all constructors for any given type will share the same `constrs` array).

We will use the following `getId()` method to extract the name of the item (block, closure entry, or top-level value) that is introduced by a given `Defn`:

```
/** Return the identifier that is associated with this definition.
 */
public String getId()
  case Defn abstract;
  case Block, ClosureDefn, TopLevel { return id; }
```

We use a similar `defines(id)` method to determine whether a given `Defn` defines a particular identifier, `id`. Note that this method ignores the (automatically generated) names of blocks and closure entries, so it will only report a match for top-level values.

```
/** Test to determine whether this Defn is for a top level value with
 *  the specified name.
 */
public boolean defines(String id)
  case Defn     { return false; }
  case TopLevel { return id.equals(this.id); }
```

Finally, we use the following macro call to introduce a class `Defns` whose values correspond to linked lists of `Defn` values. (Indeed, we have already seen one

use of a `Defns` value to record the list of entry points to a `MILProgram` in the previous section.)

```
macro List(Defn) // introduces class Defns
```

1.1.3 Monadic Code Sequences

Code sequences, which strongly resemble basic blocks in a traditional three address code representation of imperative programs, are used to describe the instructions that should be executed when a block or closure is entered. The following grammar shows a concrete syntax for code sequences:

```
c    ::=  v <- t; c        -- monadic bind
      |   t                -- tail call (ends a block)
      |   case v of alts   -- case construct (ends a block)

alts ::=  {alt1;...;altn}               -- alternatives

alt  ::=  C(v1,...,vn) -> b(a1,...,an)  -- match against C
      |   _            -> b(a1,...,an)  -- default branch
```

A code sequence of the form `v <- t ; c`, sometimes referred to as a "monadic bind", describes a computation that begins by executing/evaluating the expression `t`, binds the result to the variable `v`, and then continues to execute the rest of the code in `c`. Note that the variable `v`, recording the value produced by `t`, will be in scope (and hence can be referenced) in `c`, but it is not in scope in `t`. We will sometimes encounter code sequences of the form `v <- t ; c` where the variable `v` does not appear in `c`; in examples like this, we will often replace the variable name with an underscore/wildcard, writing `_ <- t ; c` to signal that the value produced by `t` will not be used in the subsequent computation. In general, however, we cannot just remove the initial `v <- t;` prefix of this code sequence altogether because that might change the overall semantics of the program if `t` has some potential side effect.

More generally, a complete code sequence has the form:

```
v1 <- t1; v2 <- t2; ...; vn <- tn; e
```

which is equivalent to the following notational variant, written with a more vertical layout (relying on indentation to make the use of semicolons optional after each of the separate bind expressions):

```
v1 <- t1
v2 <- t2
...
```

```
vn <- tn
e
```

In either case, running this code sequence will evaluate each of the expressions `t1`, ..., `tn` (saving the result at each step in the corresponding variable `v1`, ..., `vn`) until it reaches an end expression, `e`, which will have one of two possible forms: either an unconditional jump (i.e., a generalized "tail call", `t0`); or else a conditional jump (i.e., a `case` construct, described below). Note that, like a traditional basic block, there are no labels in the middle of code sequences. As a result, we cannot enter a code sequence part way through, and once we do enter a code sequence, we will not leave it until we reach the end expression `e`.

An end expression `case v of alts` describes a conditional jump that uses the value of the variable `v`—sometimes referred to as the "discriminant" of the case—to choose between the alternatives listed in `alts`. More specifically, a `case` construct is executed by scanning the list of alternatives from left to right and executing the block call `b(a1,...,an)` for the first alternative that matches the value of the variable `v`. For example, the following expression corresponds to a simple form of if-then-else construct that will execute `b1(x,y)` if `v` is `True()`, or `b2(z)` if `v` is `False()`.

```
case v of { True() -> b1(x,y); False() -> b2(z) }
```

We will assume that MIL programs are produced by compiling well-typed source code, so we will not worry about having to mix constructors of different types in a single list of alternatives. To put this another way, we can also write the previous expression as follows, using a default branch (signalled by the use of an underscore) in place of the previous case for `False()`.

```
case v of { True() -> b1(x,y); _ -> b2(z) }
```

Under our assumption that MIL programs are well-typed, this alternative definition is equivalent to the first because, if the variable `v` is a Boolean (as implied by the use of `True()` in the first alternative), then the only possible value for `v` that would not have been matched by the first alternative—and hence will be matched by the second—is `False()`. Once again, we will sometimes write expressions like this with a more vertical layout, eliding the semicolon and braces punctuation in the process.

```
case v of
  True() -> b1(x,y)
   _     -> b2(z)
```

In general, a `case` construct can bind variables that can appear in the block call on the right hand side of each alternative. The following group of definitions, for example, define a function for computing the length of an input list. The

`length` block enters the `loop` block with an initial (accumulating) parameter `0`. The `loop` block uses a `case` construct to examine the given `list` value. A `Nil()` value represents an empty list, in which case we just return the value that has been computed in the first argument, `n`. Otherwise, a `Cons(head,tail)` value represents a non-empty list with a first value `head`, and with the rest of the list in `tail`. In this case, we ignore the `head` value (which does not contribute to the calculation of the length) but pass the `tail` on to the `step` block, which increments the counter, and then loops back to examine the rest of the list.

```
length(list)
  = loop(0, list)

loop(n, list)
  = case list of
      Nil()           -> return n
      Cons(head,tail) -> step(n, tail)

step(n, list)
  = m <- add((n,1))
    loop(m, list)
```

We will use the classes `Bind`, `Done`, and `Match`, described below, all of which are subclasses of an abstract base class `Code` to provide the representation for MIL code sequences.

```
/** Base class for representing MIL code sequences.
 */
public abstract class Code {
  /** Represents a code sequence that binds the variable v to the result
   *  produced by running t and then continues by executing the code in c.
   */
  public case Bind(/** The variable that will capture the result.
                   */
                  private Var v,

                  /** The tail whose result will be stored in v.
                   */
                  private Tail t,

                  /** The rest of the code sequence.
                   */
                  private Code c)

  /** Represents a code sequence that just executes a single Tail.
   */
  public case Done(/** The tail to be executed.
                   */
                  private Tail t)

  /** Represents a code sequence that implements a conditional jump, using the value
   *  in a to determine which of the various alternatives in alts should be used, or
   *  taking the default branch, def, is there is no matching alternative.
   */
  public case Match(/** The discriminant for this Match.
                    */
                   private Atom a,
```

```
                    /** A list of alternatives for this Match.
                     */
                    private TAlt[] alts,

                    /** A default branch, to be used if none of the
                     *  alternatives apply.
                     */
                    private BlockCall def)
}
```

Note that the MIL `case` construct is represented by the class `Match` in the definitions above. (The name `Case` was in use for something else at the time these names were chosen!) Each `Match` includes a (possibly empty) array of alternatives, each of which is an instance of the `TAlt` class defined below. Notice also that, the `TAlt` type only represents branches of the form `C(v1,...,vn) -> b(a1,...,an)`. Instead of providing a special form of `TAlt` to describe default branches (i.e., branches of the form `_ -> b(a1,...,an)`), we include a single `def` field in each `Match` to specify a default. (There is no need to allow for more than one default because a second default would never be used. Similarly, if there are no defaults, then we can just use a value of `null` for `def`.)

```
/** Represents an alternative in a monadic Match.
 */
public class TAlt(private Cfun c, private Var[] args, private BlockCall bc)
```

One operation that we will need in the following is a way to determine whether a given variable `w` occurs free in a particular code sequence `c`. If this test succeeds, for example, then we can rewrite a code sequence of the form `(w <- t; c)` as `(_ <- t ; c)`, which could be a first step to removing the use of `t` altogether as dead code if we can be sure that it has no externally visible side effect. This operation can be implemented as follows, with appropriate cases for each of the different classes defined in this section (and the use of a corresponding `contains()` method for the `Tail` classes that are defined in the next section):

```
/** Test for a free occurrence of a particular variable.
 */
public boolean contains(Var w)
  case Code abstract;
  case Done { return t.contains(w); }
  case Bind { return t.contains(w) || (v!=w && c.contains(w)); }
  case Match {
      if (a==w || (def!=null && def.contains(w))) {
          return true;
      }
      for (int i=0; i<alts.length; i++) {
          if (alts[i].contains(w)) {
              return true;
          }
      }
      return false;
  }
  case TAlt {
      for (int i=0; i<args.length; i++) {
```

```
        if (args[i]==w) return false;
    }
    return bc.contains(w);
}
```

1.1.4 Generalized Tail Calls

The definition of MIL limits the expressions that can appear either on the right hand side of a monadic bind, or in a tail call at the end of a code sequence, to the different forms shown in the following grammar. We refer to these collectively as "tail expressions".

```
t  ::=  return a          -- monadic return
     |  p((a1, ..., an)) -- primitive call
     |  b(a1, ..., an)   -- basic block entry point
     |  C(a1, ..., an)   -- constructor (allocate boxed value)
     |  k{a1, ..., an}   -- closure (allocate closure object)
     |  f @ a            -- (first class) function application
     |  m[a1, ..., an]   -- computation (allocate thunk object)
     |  invoke a         -- invoke a thunk
```

Each of the different forms of tail expression performs some computation and returns a result.

- An expression of the form `return a` represents a computation that just returns the value of `a` immediately, with no further action. Note that the symbol `a` here represents an atom, as defined in the next section, which must be either a simple variable name or else an integer literal. As such, a `return a` expression really cannot do any real work at all, other than producing an already-computed result.

- An expression of the form `p((a1,...,an))` represents a call to a primitive function, `p`, with the specified list of arguments (all of which must be atoms). The set of primitives can be changed to suit different base environments, but typically contains at least standard arithmetic operations (such as the `add` and `mul` primitives that we assumed in the earlier examples), comparisons, builtin operations on base types, and I/O mechanisms. The representation of primitives is discussed further in Section 1.1.5. Note that we write primitive calls with a double pair of parentheses around the argument list so that they are more visibly distinguished from block calls.

- An expression of the form `b(a1, ..., an)` represents a call to a block `b` with a particular list of atoms as arguments, as has already been illustrated by the examples in previous sections.

- An expression of the form `C(a1, ..., an)` represents a call to a constructor function that allocates a data object on the heap with the list of atoms

specified as arguments as its components. Again, we have already seen several examples of this in previous sections.

- An expression of the form `k{a1, ..., an}` represents a closure constructor that allocates a closure data structure in the heap with code pointer `k` and stored free variables `a1`, ..., `an`. An expression of this form should only appear in a program that also includes a corresponding closure definition for `k` of the form described in Section 1.1.2 to describe the computation that should be performed when the closure is entered with a particular argument value.

- An expression of the form `f @ a` represents the invocation of a function (represented by a closure bound to the variable `f`) to an argument (represented by the atom `a`). In particular, if `f` holds the value `k{a1, ..., an}` and `k{x1, ..., xn} x = c`, then `f @ a` will return whatever value is produced by executing the code sequence `[a1/x1,...,an/xn]c`. (The expression `[a1/x1,...,an/xn]c` indicates the code sequence that is obtained from `c` by substituting all free occurrences of `x1` with `a1`, all free occurrences of `x2` with `a2`, and so on.)

- An expression of the form `m[a1, ..., an]` represents the allocation of a (monadic) thunk, or suspended computation. Like a closure, a thunk is a heap-allocated data structure that stores both a code pointer (`m`) and a list of values (described by the atoms `a1`, ..., `an`. Unlike a closure, however, the execution of the code for a thunk is triggered not by passing it an argument, but instead by using `invoke` expressions, described below, to force its evaluation, possibly at multiple times. In particular, thunks like this are used in the representation of monadic primitives that can have visible side effects.

- An expression of the form `invoke a` represents an invocation of the monadic thunk referenced by the atom `a`. For example, if `a` has been bound to a thunk `m[a1, ..., an]`, then the expression `invoke a` produces the same result as executing the block call `m(a1, ..., an)`.

These different forms of tail expression are represented by a collection of classes, all sharing the same abstract base class, `Tail`:

```
public abstract class Tail {   // Tail expression

    public case Return(private Atom a)               // return a
    public case Enter(private Atom f, private Atom a) // f @ a
    public case Invoke(private Atom a)               // invoke a

    public abstract case Call {
        /** The list of arguments for this call. */
        protected Atom[] args;

        /** Set the arguments for this body.  A typical pattern for constructing a
         *  Tail is:  new PrimCall(p).withArgs(args).  In this way, we can specify
         *  the arguments at the time of construction, but we also have flexibility
```

```
         *  to fix the arguments at some point after construction instead.
         */
        public Call withArgs(Atom[] args) {
            this.args = args;
            return this;
        }

        public int getArity() {
            return args.length;
        }

        public case BlockCall(private Block b)            // call a basic block
        public case PrimCall(private Prim p)              // call a primitive
        public abstract case Allocator {                  // allocators:
            public case DataAlloc(private Cfun c)         // - allocate a data value
            public case ClosAlloc(private ClosureDefn k)  // - allocate a closure
            public case CompAlloc(private Block m)        // - allocate a monadic thunk
        }
    }
}
```

The definition above also introduces two additional abstract classes that reflect similarities between different forms of tail expression. The `Call` class, for example, serves as the base for all of the tail expressions that include a list of arguments as part of their syntax. The `Allocator` class provides a base for the `DataAlloc`, `ClosAlloc`, and `CompAlloc` classes, which represent the set of all tail expressions that allocate a data structure in the heap. One reason to be interested in allocators like these as a general group is that an allocator call in a code sequence `(_ <- alloc; c)` can be deleted without changing the semantics of the program because the side effect of performing an allocation operation is not externally visible if the result is not used. (To justify this fully, we must overlook the possibility that an allocator might cause a program to fail with an out of memory error if the heap is not large enough.)

More generally, we will refer to an expression as "pure" if its execution does not have any externally visible side effects. This includes not only allocators, but also `return a` expressions and certain primitive calls. The `isPure()` method defined below is used to test for pure tail expressions, delegating to the corresponding `isPure()` method in the `Prim` class in the special case of method calls (See Section 1.1.5).

```
/** Test to determine whether a given tail expression is pure, that is,
 *  if it might have an externally visible side effect.
 */
public boolean isPure()
  case Tail      { return false; }       // Tail expressions can have effects.
  case Return    { return true; }        // But a return is always pure;
  case PrimCall  { return p.isPure(); }  // Some primitives are pure; and
  case Allocator { return true; }        // Allocators are treated as pure
```

Note that this definition for `isPure()` is overly conservative. For example, it may be possible to extend this in future to use the results of a preceding program analysis to identify pure `BlockCall`s. Of course, for a block call `b(a1, ..., an)` to be considered pure, we must ensure that the code sequence for `b` does not use any impure operations, and we must also be able to determine

that the code sequence is guaranteed to terminate. (We want to be able to treat any pure components of a program whose final values are not used as dead code; but clearly, we cannot remove a call to block that might enter an infinite loop because that could change the meaning of the program.)

To complete the definition of the `contains()` methods for code sequences that we gave in the previous section, we add a corresponding set of definitions for each of the different forms of `Tail` expression. Fortunately, the implementations in each case are straightforward:

```
/** Test to see if this Tail expression includes a free occurrence of a
 *  particular variable.
 */
public boolean contains(Var w)
  case Tail abstract;
  case Return { return a==w; }
  case Enter  { return f==w || a==w; }
  case Invoke { return a==w; }
  case Call   {
      for (int i=0; i<args.length; i++) {
          if (w==args[i]) {
              return true;
          }
      }
      return false;
  }
```

We will also define a set of methods for determining when two given `Tail` expressions are the same. For example, this operation might be useful in an implementation of common subexpression elimination: if `t1` is pure and `t1` is equal to `t2`, then a code sequence of the form `(v <- t1; w <- t2; c)` can be simplified to `(v <- t1; w <- return v; c)`, and then further simplified (using the monad laws mentioned previously) to `(v <- t1; [v/w]c)`.

The code for `sameTail()` is reasonably straightforward, but requires a fair amount of boilerplate to implement the double dispatch that is necessary to compare two arguments. The initial `sameTail()` method call in an expression of the form `t1.sameTail(t2)` distinguishes between the different cases for `t1`, but then a second method call is necessary in each case to determine whether `t2` is an expression of the same type. We handle the second dispatch in each case by creating a family of `isX()` methods, each of which is designed to compare its receiver with a specific argument value of type `X`. To speed the task of writing all of these `isX()` methods, we define a macro, `SameTail` that generates the necessary boilerplate, and just leaves us to fill in the code

```
/** Test to see if two Tail expressions are the same.
 */
public boolean sameTail(Tail that)
  case Tail abstract;
  case Return, Enter, Invoke, BlockCall, PrimCall,
       DataAlloc, ClosAlloc, CompAlloc { return that.same@class(this); }

macro SameTail(X) { boolean same\X(X that) case Tail { return false; } case X }

macro SameTail(Return)    ! { return this.a.sameAtom(that.a); }
```

```
macro SameTail(Enter)     ! { return this.f.sameAtom(that.f) && this.a.sameAtom(that.a); }
macro SameTail(Invoke)    ! { return this.a.sameAtom(that.a); }
macro SameTail(BlockCall) ! { return this.b==that.b && this.sameArgs(that); }
macro SameTail(PrimCall)  ! { return this.p==that.p && this.sameArgs(that); }
macro SameTail(DataAlloc) ! { return this.c==that.c && this.sameArgs(that); }
macro SameTail(ClosAlloc) ! { return this.k==that.k && this.sameArgs(that); }
macro SameTail(CompAlloc) ! { return this.m==that.m && this.sameArgs(that); }
```

To determine whether two `Call` expressions of the same type are equal, we need to compare the lists of arguments that are supplied in each call. We implement this using the `sameArgs()` method in the `Call` class. We assume that this method will only be used to compare two calls to the same block, primitive, etc., and hence we can assume that each of the calls has the same number of arguments. (There are no variable argument calls in MIL.)

```
/** Test to see if two Tail expressions s are the same.
 */
public boolean sameArgs(Call that)
  case Call {
      for (int i=0; i<args.length; i++) {
          if (!this.args[i].sameAtom(that.args[i])) {
              return false;
          }
      }
      return true;
  }
```

1.1.5 Primitives

MIL programs use primitive calls (values of type `PrimCall`) to handle basic computations for manipulating values of builtin types such as arithmetic operations, comparisons, and so on. Each primitive call specifies a particular primitive operation, represented by a value of the following `Prim` class, that specifies the name of the primitive, its arity (i.e., the number of arguments that are required in any `PrimCall` that uses this primitive), and a `purity` field to specify what kind of externally visible effect (if any) might occur as a result of using the primitive. The possible values of `purity` are `PURE`, `IMPURE`, and `THUNK`, each of which is described further in the comments below.

```
public class Prim(/** The name that will be used for this primitive.
                   */
                  private String id,

                  /** The arity/number of arguments for this primitive.
                   */
                  private int arity,

                  /** Records the purity setting (PURE, IMPURE, or THUNK) for this
                   *  primitive.
                   */
                  private int purity,

                  /** Flag to indicate if this function will never return,
                   *  meaning that anything following a call to this function
                   *  will be ignored.
```

```
                 */
                private boolean doesntReturn) {

  /** Return the name of this primitive.
   */
  public getter id;

  /** Return the arity for this primitive.
   */
  public getter arity;

  /** Identifies a "pure" primitive, that is, a primitive that has no effect other than to
   *  compute a value.  In particular, a call to a pure primitive in a context where the
   *  result will not be used can always be deleted without a change in program semantics.
   */
  public static final int PURE   = 0;

  /** Identifies an "impure" primitive, which is a primitive that, although not monadic,
   *  can have an effect when executed.  In particular, this means that a call to an
   *  impure primitive cannot be deleted, even if its result is not used.  This includes
   *  operations like division that, without any typing tricks to eliminate division by
   *  zero, can raise an exception.  Removing a use of division---except in cases where
   *  the second argument is a known, nonzero constant---could potentially change the
   *  meaning of a program.
   */
  public static final int IMPURE = 1;

  /** Identifies a "monadic" primitive that requires a thunk.
   */
  public static final int THUNK  = 2;

  /** Determine if this primitive is pure, i.e., if a call to this primitive can
   *  be deleted if its result is not used.
   */
  public boolean isPure() { return purity==PURE; }
}
```

For convenience, we define the following values of type `Prim` to represent some of the most commonly used primitives for working with integer values in MIL. All but one of these are treated as `PURE` operations, the exception being integer division, which can potentially trigger an externally visible divide by zero exception if its second argument is zero.

```
class Prim {
  // Bitwise operators (on integers):
  public static final Prim not  = new Prim("not",  1, PURE, false);
  public static final Prim and  = new Prim("and",  2, PURE, false);
  public static final Prim or   = new Prim("or",   2, PURE, false);
  public static final Prim xor  = new Prim("xor",  2, PURE, false);

  // Boolean negation:
  public static final Prim bnot = new Prim("bnot", 1, PURE, false);

  // Shift operations:
  public static final Prim shl  = new Prim("shl",  2, PURE, false);
  public static final Prim shr  = new Prim("shr",  2, PURE, false);

  // Integer arithmetic:
  public static final Prim neg  = new Prim("neg",  1, PURE, false);
  public static final Prim add  = new Prim("add",  2, PURE, false);
  public static final Prim sub  = new Prim("sub",  2, PURE, false);
  public static final Prim mul  = new Prim("mul",  2, PURE, false);
  public static final Prim div  = new Prim("div",  2, IMPURE, false);
```

```
  // Integer comparisons:
  public static final Prim eq   = new Prim("eq",   2, PURE, false);
  public static final Prim neq  = new Prim("neq",  2, PURE, false);
  public static final Prim lt   = new Prim("lt",   2, PURE, false);
  public static final Prim lte  = new Prim("lte",  2, PURE, false);
  public static final Prim gt   = new Prim("gt",   2, PURE, false);
  public static final Prim gte  = new Prim("gte",  2, PURE, false);
}
```

As a special case, we define a primitive, together with corresponding `PrimCall` and `Code` values, called `halt`, to represent an operation that will terminate or abort the current program if it is executed. Note that this primitive requires the `IMPURE` setting (because deleting a call to `halt` could certainly change the behavior of the program), but can also set `doesntReturn` to `true` because code that follows a `halt` will never be executed.

```
class Prim {
  /** Represents a primitive that halts/terminates the current program.
   */
  public static final Prim halt = new Prim("halt", 0, IMPURE, true); // halt/abort
}
class PrimCall {
  /** Represents a tail expression that halts/terminates the current program.
   */
  public static final Call halt = new PrimCall(Prim.halt).withArgs(new Atom[0]);
}
class Code {
  /** Represents a code sequence that halts/terminates the current program.
   */
  public static final Code halt = new Done(PrimCall.halt);
}
```

None of the examples listed here are `THUNK` operations; that feature is only needed when we are using MIL code to implement a purely functional programming language where the side-effecting implementations of monadic primitives must be wrapped up inside thunks.

It is easy to introduce additional primitives by declaring new values of type `Prim`. Of course, none of these definitions really says very much about what each primitive actually does. We have assumed that the intended meaning for each of the primitives declared above is fairly obvious from the name and arity values that we have chosen in each case, but of course some additional code will be needed to provide the appropriate implementations.

1.1.6 Atoms

Atoms are either variable names or integer literals, as described by the following grammar:

```
a  ::=  v     -- variable
     |  n     -- an unboxed constant (i.e., an integer literal)
```

These are the simplest forms of expression in MIL programs, and they correspond to register or immediate addressing modes in a typical assembly programming language. Atoms are the only form of expression that can be used as arguments in `Call` expressions (as described in the previous section), which forces us to break down the code for computing the value of a large, compound expression in to a sequence of individual bindings, each of which performs a single operation.

We represent different forms of atoms as values of the following `Atom` class, with distinct subclasses `Var` for variables and `Const` for integer constants. Notice, however, that `Var` is also defined as an abstract class, and we define some further subclasses to distinguish between references to top-level definitions (class `Top`), compiler-generated temporaries (class `Temp`), and wildcard variables (class `Wildcard`) that are used in situations where the value produced by a computation will not actually be used in the rest of the program. Note that we can construct two different forms of temporary variable: a call of the form `new Temp(id)` creates a variable with the specified name, `id`, which might, for example, correspond to the name of a function parameter or local variable in a source program; however, a call of the form `new Temp()` creates a variable with a compiler-generated name, each of which begins with the letter `t` followed by a numeric suffix.

```
/** Represents basic atoms in a MIL program, each of which is either a
 *  variable or an integer literal.
 */
public abstract class Atom {

  public abstract case Var {
    public static final Var[] noVars = new Var[0];

    public case Top(private TopLevel tl)     // reference to top-level variable
    public case Temp(private String id) {    // temporary variable
        private static int count = 0;
        public Temp() {
            this("t"+count++);
        }
    }
    public case Wildcard                     // unreferenced variable
  }

  public case Const(private int val) {
      public getter val;
  }
}
macro Singleton(Wildcard)
```

It is important to understand that the strings that we store in different types of `Var` object are only used for displaying textual descriptions of MIL programs, as in the following implementation of `toString()`, which can be used to generate an appropriate `String` value for each different `Atom` value:

```
/** Generate a printable description of this atom.
 */
public String toString()
  case Atom     abstract;
```

```
  case Wildcard { return "_"; }
  case Top      { return tl.getId(); }
  case Temp     { return id; }
  case Const    { return "" + val; }
```

If, instead, we want to compare two values of type `Var` for equality, then we will ignore the `String` values and instead just use pointer equality to see if the two `Var` objects are the same. In particular, if a program contains distinct uses of a single variable name, then each of those uses should be represented by a different `Var` object. (In addition, this also means that we will not confuse compiler-generated temporaries with top-level or user-specified temporaries, even if they are displayed in the same way.)

The following code uses these ideas to define a `sameAtom()` method for determining whether two atoms refer to the same item:

```
/** Test to see if two atoms are the same.  For a pair of Const objects,
 *  this means that the two objects have the same val.  For any other
 *  pair of Atoms, we expect the objects themselves to be the same.
 */
public boolean sameAtom(Atom that)
  case Atom  { return this==that; }
  case Const {
      Const c = that.isConst();
      return c!=null && c.getVal()==this.val;
  }

/** Test to determine whether this Atom is a constant (or not).
 */
public Const isConst()
  case Atom  { return null; }
  case Const { return this; }
```

The final pair of method implementations in this section are utilities for constructing a new array of temporary variables, either for an array containing the individual identifier names that should be used, or else just by specifying the number of new arguments that are required.

```
class Temp {
  /** Create a list of new variables corresponding to a given list of identifiers.
   */
  public static Var[] makeTemps(String[] ids) {
      int   n  = ids.length;
      Var[] vs = new Var[n];
      for (int i=0; i<n; i++) {
          vs[i] = new Temp(ids[i]);
      }
      return vs;
  }

  /** Create a list of new variables of a given length.
   */
  public static Var[] makeTemps(int n) {
      Var[] vs = new Var[n];
      for (int i=0; i<n; i++) {
          vs[i] = new Temp();
      }
      return vs;
  }
}
```

1.2 Dependency Analysis

In this section, we describe an implementation of dependency analysis for MIL programs. This process starts by building a graph structure that captures the dependencies between the definitions that make up a given program, and then performs a strongly-connected components analysis of the resulting graph. Each of the strongly-connected components (SCCs) that are produced by this analysis contains either a single, non-recursive definition, or else a small group of one or more mutually recursive definitions, all of which must typically be considered together in the process of further analyzing or optimizing a given MIL program. As an important added benefit, because the analysis only examines definitions that are reachable from one or more of the program's entry points, we know that the resulting set of SCCs will only contain definitions that are referenced, either directly or indirectly from those same entry points. In effect, this performs a form of tree shaking or dead code elimination that automatically removes parts of the program that are not required to run it.

Our implementation of dependency analysis leverages some reusable infrastructure, detailed in Appendix A, for computing SCCs.

```
// We will want to do SCC analysis on lists of definitions:
macro SCC(Defn)     // adds callees and callers fields to each Defn
macro AddIsIn(Defn)
// TODO: We don't need the forward search pieces of SCC for Defns

class Defn {
    private static int dfsNum = 0;   // Number of current depth-first search
    private int visitNum = 0;        // Number of most recent search

    public static void newDFS() {    // Begin a new depth-first search
        dfsNum++;
    }

    protected int occurs;            // Count total occurrences
    public getter occurs;

    /** Visit this Defn as part of a depth first search, and building up a list
     *  of Defn nodes that could be used to compute strongly-connected components.
     */
    Defns visitDepends(Defns defns) {
        if (visitNum==dfsNum) {        // Repeat visit to this Defn?
            occurs++;
        } else {                       // First time at this Defn
            // Mark this Defn as visited, and initialize fields
            visitNum = dfsNum;
            occurs   = 1;
            scc      = null;
            callers  = null;
            callees  = null;

            // Find immediate dependencies
            Defns deps = dependencies();

            // Visit all the immediate dependencies
            for (; deps!=null; deps=deps.next) {
                defns = deps.head.visitDepends(defns);
                if (!Defns.isIn(deps.head, callees)) {
                    callees = new Defns(deps.head, callees);
                }
```

```
            }

            // Add the information about this node's callers/callees
            this.calls(callees);
            // And add it to the list of all definitions.
            defns = new Defns(this, defns);
        }
        return defns;
    }
}

/** Find the list of Defns that this Defn depends on.
 */
public Defns dependencies()
  case Defn abstract;
  case Block       {  // b(args) = code
      return code.dependencies(null);
  }
  case ClosureDefn {  // k{stored} arg = tail
      return tail.dependencies(null);
  }
  case TopLevel    {  // id <- tail
      return tail.dependencies(null);
  }

/** Find the list of Defns that this Code sequence depends on.
 */
public Defns dependencies(Defns ds)
  case Code abstract;
  case Bind {  // v <- t; c
      return t.dependencies(c.dependencies(ds));
  }
  case Done {  // t
      return t.dependencies(ds);
  }
  case Match { // case a of alts; d
      if (def!=null) {
          ds = def.dependencies(ds);
      }
      for (int i=0; i<alts.length; i++) {
          ds = alts[i].dependencies(ds);
      }
      return a.dependencies(ds);
  }
  case TAlt {  // c args -> bc
      return bc.dependencies(ds);
  }

/** Find the list of Defns that this Tail depends on.
 */
public Defns dependencies(Defns ds)
  case Tail abstract;
  case Return    { // a
      return a.dependencies(ds);
  }
  case Enter     { // f @ a
      return f.dependencies(a.dependencies(ds));
  }
  case Invoke    { // invoke a
      return a.dependencies(ds);
  }
  case BlockCall { // b(args)
      return b.dependencies(Atom.dependencies(args, ds));
  }
  case PrimCall  { // p(args)
      return Atom.dependencies(args, ds); // TODO: do we need to register prims?
  }
  case DataAlloc { // c(args)
```

```
      return Atom.dependencies(args, ds); // TODO: do we need to register cfuns?
  }
  case ClosAlloc { // k{args}
      return k.dependencies(Atom.dependencies(args, ds));
  }
  case CompAlloc { // m[args]
      return m.dependencies(Atom.dependencies(args, ds));
  }
  case Atom {
      return ds;     // Only Top atoms can register
  }
  case Top {
      return tl.dependencies(ds);
  }
  case Defn {
      return /* Defns.isIn(this, ds) ? ds : */ new Defns(this, ds); // count multiplicities
  }

class Atom {
  /** Find out the list of Defns that this array of Atoms depends on.
   */
  public static Defns dependencies(Atom[] args, Defns ds) {
      if (args!=null) {
          for (int i=0; i<args.length; i++) {
              ds = args[i].dependencies(ds);
          }
      }
      return ds;
  }
}

class MILProgram {
  /** Record the list of strongly connected components in this program.
   */
  private DefnSCCs sccs;

  /** Compute the list of definitions for the reachable portion of the input graph.
   */
  public Defns reachable() {
      Defn.newDFS();                      // Begin a new depth-first search
      Defns defns = null;                 // Compute a list of reachable Defns
      for (Defns ds=entries; ds!=null; ds=ds.next) {
          defns = ds.head.visitDepends(defns);
      }
      if (defns==null) {
          System.out.println("No definitions remain");
      }

      return defns;
  }

  /** Perform tree shaking on this program, computing strongly-connected components
   *  for the reachable portion of the input graph.
   */
  public void shake() {
      sccs = Defns.searchReverse(reachable()); // Compute the strongly-connected components
  }
}

/*
void describe() case Defn {
    System.out.print(getId() + "::  calls ");
    String msg = "";
    for (Defns cs=callers; cs!=null; cs=cs.next) {
        System.out.print(msg); msg = ", ";
        System.out.print(cs.head.getId());
    }
    System.out.print("; callees "); msg = "";
```

```
    for (Defns cs=callees; cs!=null; cs=cs.next) {
        System.out.print(msg); msg = ", ";
        System.out.print(cs.head.getId());
    }
    System.out.println(".");
}
*/
```

1.2.1 Display Results in Graphviz Format

```
class MILProgram {
  import java.io.PrintWriter;
  import java.io.BufferedWriter;
  import java.io.FileWriter;
  import java.io.IOException;

  public void toDot(String name) {
      try {
          PrintWriter out = new PrintWriter(new BufferedWriter(new FileWriter(name)));
          out.println("digraph MIL {");
          for (Defns es=entries; es!=null; es=es.next) {
              out.println("  " + es.head.getId() + ";");
          }
          for (Defns ds=reachable(); ds!=null; ds=ds.next) {
              for (Defns cs=ds.head.getCallers(); cs!=null; cs=cs.next) {
                  out.println("  " + cs.head.getId() + " -> " + ds.head.getId() + ";");
              }
          }
          out.println("}");
          out.close();
      } catch (IOException e) {
          System.out.println("Attempt to create dot output file " + name + " failed");
      }
  }
}

class Defn { public getter callers; }
```

1.3 Display MIL Programs

As a first exercise in using the abstract syntax classes, this section provides code for displaying MIL programs, and the associated `Code` and `Tail` structures from which they are built, in a readable concrete syntax. These functions are particularly useful for debugging, or for otherwise inspecting the output of the compiler and optimizer that will be described in later chapters. The syntax that we use, as well as the code for producing it, is mostly straightforward, with a distinct case for each of the main constructs in MIL:

```
macro AddLength(Defns)  // add implementation of length on Defns

/** Display a printable representation of this MIL construct
 *  on the standard output.
 */
public void display()
  case MILProgram {
      for (DefnSCCs dsccs = sccs; dsccs!=null; dsccs=dsccs.next) {
          dsccs.head.display();
```

```
      }
  }
  case DefnSCC {
      System.out.println("-------------------------------------");
      System.out.println(isRecursive() ? "recursive" : "not recursive");
      for (Defns ds=bindings; ds!=null; ds=ds.next) {
        ds.head.display();
      }
  }
  case Defn {
      displayDefn();
  }

  case Code abstract;
  case Bind {
      indent();
// TODO: hiding the _ <- part of a bind seems mostly like a good thing to do,
//       but it does look a bit confusing when the right hand side expression
//       is a return ... revisit this, but for now I'm keeping the _ <- part.
//    if (v!=Wildcard.obj) {
      System.out.print(v.toString());
      System.out.print(" <- ");
//    }
      t.displayln();
      c.display();
  }
  case Done {
      indent();
      t.displayln();
  }
  case Match {
      indentln("case " + a + " of");
      for (int i=0; i<alts.length; i++) {
          indent();  // double indent
          indent();
          alts[i].display();
      }
      if (def!=null) {
          indent();  // double indent
          indent();
          System.out.print("_ -> ");
          def.displayln();
      }
  }
  case TAlt {
      Call.display(c.getId(), "(", args, ")");
      System.out.print(" -> ");
      bc.displayln();
  }

  case Tail abstract;
  case Return    { System.out.print("return " + a); }
  case Enter     { System.out.print(f + " @ " + a); }
  case Invoke    { System.out.print("invoke " + a); }
  case BlockCall { display(b.getId(), "(",  args, ")");  }
  case PrimCall  { display(p.getId(), "((", args, "))"); }
  case DataAlloc { display(c.getId(), "(",  args, ")");  }
  case ClosAlloc { display(k.getId(), "{",  args, "}");  }
  case CompAlloc { display(m.getId(), "[",  args, "]");  }

void displayDefn()
  case Defn abstract;
  case Block {
//System.out.println("doesntReturn = " + doesntReturn);
      Call.display(id, "(", formals, ")");
      System.out.println(" =");
      if (code==null) {
          Code.indent();
```

```
            System.out.println("null");
        } else {
            code.display();
        }
    }
    case ClosureDefn {
        Call.display(id, "{", stored, "} ");
        System.out.print(arg + " = ");
        tail.displayln();
    }
    case TopLevel {
        System.out.print(id + " <- ");
        if (tail==null) {
            System.out.println("null");
        } else {
            System.out.println();
            Code.indent();
            tail.displayln();
        }
    }
```

The remaining code provides simple auxiliary functions that are used in the code above:

```
/** Display a Tail value and then move to the next line.
 */
public void displayln()
  case Tail { display(); System.out.println(); }

class Code {
  /** Print an indent at the beginning of a line.
   */
  public final static void indent() {
      System.out.print("  ");
  }

  /** Print a suitably indented string at the start of a line.
   */
  public final static void indentln(String s) {
      indent();
      System.out.println(s);
  }
}

class Call {
  /** Print a call with a notation that includes the name of the item that is being
   *  called and a list of arguments, appropriately wrapped between a given open and
   *  close symbol (parentheses for block, primitive, and data constructor calls;
   *  braces for closure constructors; and brackets for monadic thunk constructors).
   */
  public static void display(String name, String open, Atom[] args, String close) {
      System.out.print(name);
      System.out.print(open);
      if (args!=null && args.length>0) {
          System.out.print(args[0].toString());
          for (int i=1; i<args.length; i++) {
              System.out.print(", ");
              System.out.print(args[i].toString());
          }
      }
      System.out.print(close);
  }
}
```

1.4 Lists of Variables

This section defines some functions for working with lists of `Var` values, represented by the `Vars` class, including standard functions for computing list length and testing for membership:

```
macro List(Var)       // introduces class Vars
macro AddIsIn(Var)    // adds support for isIn() on Vars lists
macro AddLength(Vars) // adds support for length() on Vars lists
```

To help in debugging, we include a function for producing a string that displays the variables in a given `Vars` list using standard set notation:

```
class Vars {
  public static String toString(Vars vs) {
      StringBuffer b = new StringBuffer("{");
      if (vs!=null) {
          b.append(vs.head.toString());
          while ((vs=vs.next)!=null) {
              b.append(",");
              b.append(vs.head.toString());
          }
      }
      b.append("}");
      return b.toString();
  }
}
```

We also provide a function for capturing a list of variables as an array:

```
class Vars {
  public static Var[] toArray(Vars vs) {
      Var[] va = new Var[Vars.length(vs)];
      for (int i=0; vs!=null; vs=vs.next) {
          va[i++] = vs.head;
      }
      return va;
  }
}
```

We also provide functions for adding items to a list of variables. These operations are designed to be used with calls of the form `vs = Vars.add(newVs, vs)`, where `vs` is the list of variables that is being added to and `newVs` specifies the variables that we want to add. There are variations of `add` for the cases where `newVs` is a single variable, an array, or a list of variables, as well as a convenience operator that allows us to add a single variable `v` using an expression of the form `v.addTo(vs)`.

```
class Vars {
  // Adding variables to a list: ----------------------------------
  // (destructively modifies the second argument)

  public static Vars add(Var v, Vars vs) {
      return Vars.isIn(v,vs) ? vs : new Vars(v, vs);
  }
```

```
  public static Vars add(Var[] vararray, Vars vs) {
      for (int i=0; i<vararray.length; i++) {
          vs = add(vararray[i], vs);
      }
      return vs;
  }

  public static Vars add(Vars us, Vars vs) {
      for (; us!=null; us=us.next) {
          vs = add(us.head, vs);
      }
      return vs;
  }
}

class Var {
    public Vars addTo(Vars vs) { return Vars.add(this, vs); }
}
```

A natural extension allows us to compute the set of variables that appear free (and not defined at the top-level) in a given `Tail` value.

```
/** Add the variables mentioned in this tail to the given list of variables.
 */
public Vars add(Vars vs)
  case Tail   abstract;
  case Return { return a.add(vs); }          // a
  case Enter  { return f.add(a.add(vs)); }   // f @ a
  case Invoke { return a.add(vs); }          // invoke a
  case Call   { return Vars.add(args, vs); } // body

/** Add this atom as an argument variable to the given list; only local
 *  variables and temporaries are treated as argument variables because
 *  wildcards are ignored and all other atoms can be accessed as constants.
 */
Vars add(Vars vs)
  case Atom { return vs; }
  case Temp { return Vars.add(this, vs); }

class Vars {
  /** Add the arguments in an array of atoms to a given list of variables.
   */
  public static Vars add(Atom[] args, Vars vs) {
      for (int i=0; i<args.length; i++) {
          vs = args[i].add(vs);
      }
      return vs;
  }
}
```

We also provide a corresponding set of functions for removing items from a list of variables. These operations are designed to be used with calls of the form `vs = Vars.remove(remVs, vs)`, where `vs` is the list of variables that is being modified and `remVs` specifies the variables that we want to remove. There are variations of `remove` for the cases where `remVs` is a single variable, an array, or a list of variables, as well as a convenience operator that allows us to remove a single variable `v` using an expression of the form `v.removeFrom(vs)`.

```
class Vars {
  // Removing variables from a list: (Destructive) -----------------
```

```
  // (destructively modifies the second argument)

  public static Vars remove(Var v, Vars vs) {
      Vars prev = null;
      for (Vars us=vs; us!=null; us=us.next) {
          if (us.head==v) {
              if (prev==null) {
                  return us.next;       // remove first element
              } else {
                  prev.next = us.next;  // remove later element
                  return vs;            // and return modified list
              }
          }
          prev = us;
      }
      return vs;  // variable not listed
  }

  public static Vars remove(Var[] vararray, Vars vs) {
      for (int i=0; i<vararray.length; i++) {
          vs = remove(vararray[i], vs);
      }
      return vs;
  }

  public static Vars remove(Vars us, Vars vs) {
      for (; us!=null; us=us.next) {
          vs = remove(us.head, vs);
      }
      return vs;
  }
}

class Var {
    public Vars removeFrom(Vars vs) { return Vars.remove(this, vs); }
}
```

1.5 Variable to Atom Substitutions

1.5.1 Representing and Constructing Substitutions

```
/** AtomSubst value represent substitutions of Atoms for Vars as simple
 *  linked list structures.
 */
public class AtomSubst(private Var v, private Atom a, private AtomSubst rest) {

  /** Extend a substitution with bindings given by a pair of arrays.
   */
  public static AtomSubst extend(Var[] vs, Atom[] as, AtomSubst s) {
      if (vs.length != as.length) {
          debug.Internal.error("AtomSubst.extend: variable/atom counts do not match.");
      }
      for (int i=0; i<as.length; i++) {
          s = new AtomSubst(vs[i], as[i], s);
      }
      return s;
  }
}

class AtomSubst {
  /** Remove any previous binding for the variable from the
   *  given substitution, using destructive updates.
   *
```

```
   *  TODO: is this sufficient if one of the remaining bindings still mentions w?
   */
  public static AtomSubst remove(Var w, AtomSubst s) {
      if (s==null) {
          return null;
      } else if (s.v==w) {
          return s.rest;
      } else {
          s.rest = remove(w, s.rest);
          return s;
      }
  }
}
```

1.5.2 Applying Substitutions to Atoms

```
/** Apply an AtomSubst to this atom.
 */
public Atom apply(AtomSubst s)
  case Atom   { return this; }
  case Var    { return AtomSubst.apply(this, s); }

class AtomSubst {
  /** Apply the given substitution to the specified variable.
   */
  public static Atom apply(Var w, AtomSubst s) {
      for (; s!=null; s=s.rest) {
          if (s.v==w) {
              return s.a;
          }
      }
      return w;
  }

  /** Apply the given substitution to a vector of atoms.
   */
  public static Atom[] apply(Atom[] args, AtomSubst s) {
      int    n     = args.length;
      Atom[] nargs = new Atom[n];
      for (int i=0; i<n; i++) {
          nargs[i] = args[i].apply(s);
      }
      return nargs;
  }
}
```

1.5.3 Applying Substitutions to Tail Values

```
/** Apply an AtomSubst to this Tail, skipping the operation if
 *  the substition is empty as an attempt at an optimization.
 */
public Tail apply(AtomSubst s)
  case Tail   { return (s==null) ? this : forceApply(s); }

/** Apply an AtomSubst to this Tail.
 */
public Tail forceApply(AtomSubst s)
  case Tail   abstract;
  case Return { return new Return(a.apply(s)); }
  case Enter  { return new Enter(f.apply(s), a.apply(s)); }
  case Invoke { return new Invoke(a.apply(s)); }
  case Call   { return callDup().withArgs(AtomSubst.apply(args, s)); }
```

```
/** Construct a new Call value that is based on the receiver,
 *  without copying the arguments.
 */
Call callDup()
  case Call      abstract;
  case BlockCall { return new BlockCall(b); }
  case PrimCall  { return new PrimCall(p);  }
  case DataAlloc { return new DataAlloc(c); }
  case ClosAlloc { return new ClosAlloc(k); }
  case CompAlloc { return new CompAlloc(m); }

/** A special version of apply that works only on Call values; used
 *  in places where type preservation of a Call value is required.
 *  TODO: is this still used?
 */
public Call forceApplyCall(AtomSubst s)
  case Call { return callDup().withArgs(AtomSubst.apply(args, s)); }

/** A special version of apply that works only on BlockCalls;
 *  used in places where type preservation of a BlockCall argument
 *  is required.  In particular, we don't use withArgs here because
 *  that loses type information, producing a Body from a BlockCall
 *  input.
 */
public BlockCall applyBlockCall(AtomSubst s)
  case BlockCall { return (s==null) ? this : forceApplyBlockCall(s); }

/** A special version of forceApply that works only on BlockCalls.
 */
BlockCall forceApplyBlockCall(AtomSubst s)
  case BlockCall {
      BlockCall bc = new BlockCall(b);
      bc.args      = AtomSubst.apply(args, s);
      return bc;
  }
```

1.5.4 Applying Substitutions to Code Values

```
/** Apply an AtomSubst to this Code, skipping the operation if
 *  the substitution is empty as an attempt at an optimization.
 *  This operation essentially builds a fresh copy of the original
 *  code sequence, introducing new temporaries in place of any
 *  variables introduced by Binds.
 */
public Code apply(AtomSubst s)
  case Code { return (s==null) ? this : forceApply(s); }

/** Apply an AtomSubst to this Code.
 */
public Code forceApply(AtomSubst s)
  case Code abstract;
  case Bind {    // v <- t; c
      Temp w = new Temp();
      return new Bind(w, t.forceApply(s), c.forceApply(new AtomSubst(v, w, s)));
  }
  case Done {    // t
      return new Done(t.forceApply(s));
  }
  case Match {   // case a of alts; d
      TAlt[] talts = new TAlt[alts.length];
      for (int i=0; i<alts.length; i++) {
          talts[i] = alts[i].forceApply(s);
      }
      BlockCall d = (def==null) ? null : def.forceApplyBlockCall(s);
      return new Match(a.apply(s), talts, d);
```

```
  }

/** Apply an AtomSubst to this TAlt, skipping if the substitution is empty.
 */
public TAlt apply(AtomSubst s)
  case TAlt { return (s==null) ? this : forceApply(s); }

/** Apply an AtomSubst to this TAlt.
 */
public TAlt forceApply(AtomSubst s)
  case TAlt {  // c args -> bc
/*
      // Extends substitution with fresh bindings for the variables in
      // args.  Simple, but often unnecessary allocation.
      Var[] vs = Temp.makeTemps(args.length);
      s        = AtomSubst.extend(args, vs, s);
      return new TAlt(c, vs, bc.forceApplyBlockCall(s));
*/
      // Extends substitution with identity bindings for the variables in
      // args.  Simple, but often unnecessary allocation.
      s = AtomSubst.extend(args, args, s);
      return new TAlt(c, args, bc.forceApplyBlockCall(s));
  }
```

1.6 An interpreter for MIL programs

In this section, we describe a very simple interpreter for MIL programs. This might be used for running simple tests, or perhaps as a starting point for constructing a more formal semantics of the MIL language, but it is not intended as a practical tool. Among other shortcomings, it currently does not provide implementations for most of the primitives that were described in Section 1.1.5 (although adding the code for that would not be difficult). Perhaps more seriously, the implementation relies heavily on recursion and, because Java does not support tail recursion optimization, it will likely cause stack overflows for nontrivial programs.

We begin by defining a `Fail` class that will be used for representing and reporting errors that are detected while the interpreter is running:

```
public class Fail(private String msg) extends Exception {
  public String toString() {
      return "Interpreter error: " + msg;
  }
}
```

1.6.1 Representing MIL values

Our next task is to provide a representation for the different types of value that are manipulated by MIL programs. We begin with an abstract base class, `Val`, with subclasses for integer values (`IntVal`) and heap-allocated objects (`AllocVal`). A further set of subclasses are used to distinguish between heap-allocated data values (`DataVal`, corresponding to values produced by MIL expressions of the form `C(a1,...,an)`); closure values (`ClosVal`, corresponding

to values produced by MIL expressions of the form `k{a1,...,an}`); and computation or thunk values (`CompVal`, corresponding to values produced by MIL expressions of the form `m[a1,...,an]`). In each of these three cases, we include an array, `vals`, to hold the actual values that were used for the formal parameters `a1`, ..., `an`, together with a tag to specify the appropriate constructor function, closure entry, or block entry.

```
public abstract class Val {
  case IntVal(private int num)
  abstract case AllocVal(protected Val[] vals) {
      case DataVal(private Cfun c)
      case ClosVal(private ClosureDefn k)
      case CompVal(private Block m)
  }
}
```

We define To allow us to inspect the results of the For use in debugging,

```
public String toString()
  case Val {
      StringBuffer buf = new StringBuffer();
      this.append(buf);
      return buf.toString();
  }

void append(StringBuffer buf)
  case Val      abstract;
  case IntVal  { buf.append(num); }
  case DataVal { c.append(buf); append(buf, '(', ')'); }
  case ClosVal { k.append(buf); append(buf, '{', '}'); }
  case CompVal { m.append(buf); append(buf, '[', ']'); }
  case Defn    { buf.append(getId()); }

protected void append(StringBuffer buf, char open, char close)
  case AllocVal {
      buf.append(open);
      String sep = "";
      for (int i=0; i<vals.length; i++) {
          buf.append(sep);
          sep = ", ";
          vals[i].append(buf);
      }
      buf.append(close);
  }
```

1.6.2 Value Environments

```
public class ValEnv(private Var v, private Val val, private ValEnv rest) {
  public static ValEnv extend(Var[] vars, Val[] vals, ValEnv env) {
      for (int i=0; i<vars.length; i++) {
          env = new ValEnv(vars[i], vals[i], env);
      }
      return env;
  }

  public static Val lookup(Var v, ValEnv env) throws Fail {
      while (env!=null) {
          if (v==env.v) {
              return env.val;
          }
```

```
      }
      throw new Fail("Could not find value for variable " + v);
  }

  public static Val[] lookup(Atom[] atoms, ValEnv env) throws Fail {
      Val[] vals = new Val[atoms.length];
      for (int i=0; i<atoms.length; i++) {
          vals[i] = atoms[i].lookup(env);
      }
      return vals;
  }
}

public Val lookup(ValEnv env) throws Fail
  case Atom  abstract;
  case Const { return new IntVal(val); }
  case Top   { throw new Fail("lookup Top not implemented yet"); } // TODO: Fix this!
  case Var   { return ValEnv.lookup(this, env); }
```

1.6.3 Evaluator Methods

```
//- Main Evaluator: ----------------------------------------
// Every execution step pushes us to another level of recursion;
// we really need a tail recursive implementation of Java to make
// this work in practice.

public Val eval(ValEnv env) throws Fail
  case Code      abstract;
  case Done      { return t.eval(env); }
  case Bind      { return c.eval(new ValEnv(v, t.eval(env), env)); }
  case Match     { return a.lookup(env).match(alts, def, env); }

  case Tail      abstract;
  case Return    { return a.lookup(env); }
  case Enter     { return f.lookup(env).enter(a.lookup(env)); }
  case Invoke    { return a.lookup(env).invoke(); }
  case BlockCall { return b.call(ValEnv.lookup(args, env)); }
  case PrimCall  { return p.call(ValEnv.lookup(args, env)); }
  case DataAlloc { return new DataVal(ValEnv.lookup(args, env), c); }
  case ClosAlloc { return new ClosVal(ValEnv.lookup(args, env), k); }
  case CompAlloc { return new CompVal(ValEnv.lookup(args, env), m); }

public Val match(TAlt[] alts, BlockCall def, ValEnv env) throws Fail
  case Val      { throw new Fail("Runtime type error in pattern match"); }
  case DataVal  {
      for (int i=0; i<alts.length; i++) {
          Val val = alts[i].match(c, vals, env);
          if (val!=null) {
              return val;
          }
      }
      return def.eval(env);
  }

public Val match(Cfun c, Val[] vals, ValEnv env) throws Fail
  case TAlt {
      return (c==this.c) ? bc.eval(ValEnv.extend(args, vals, env)) : null;
  }

public Val enter(Val val) throws Fail
  case Val      { throw new Fail("Runtime type error in enter"); }
  case ClosVal  { return k.enter(vals, val); }

public Val enter(Val[] vals, Val val) throws Fail
  case ClosureDefn {
```

```
      return tail.eval(new ValEnv(arg, val, ValEnv.extend(stored, vals, null)));
  }

public Val invoke() throws Fail
  case Val      { throw new Fail("Runtime type error in invoke"); }
  case CompVal  { return m.call(vals); }

public Val call(Val[] vals) throws Fail
  case Block { return code.eval(ValEnv.extend(formals, vals, null)); }
  case Prim  {
      if (arity!=vals.length) {
          throw new Fail("primitive " + id + " does not have " + arity + " arguments");
      }
      if (id.equals("add")) {
          return new IntVal(vals[0].asInt() + vals[1].asInt());
      } else if (id.equals("sub")) {
          return new IntVal(vals[0].asInt() - vals[1].asInt());
      } else if (id.equals("eq")) {
          return Val.asBool(vals[0].asInt() == vals[1].asInt());
      } else {
        throw new Fail("No implementation for primitive " + id);
      }
  }

public int asInt() throws Fail
  case Val    { throw new Fail("Runtime type error, integer value expected"); }
  case IntVal { return num; }

class Val {
  public static final Val[] noVals = new Val[0];
  public static final Val trueVal  = new DataVal(noVals, Cfun.True);
  public static final Val falseVal = new DataVal(noVals, Cfun.False);
  public static Val asBool(boolean b) {
      return b ? trueVal : falseVal;
  }
}
```

Chapter 2

Optimization

This chapter presents algorithms for transforming MIL code in ways that are intended to improve the performance of compiled code, typically by reducing run-time overhead, eliminating redundant computations, or switching to more efficient methods for computing particular values. The following list provides an overview of the main techniques, glossing over technical details that will be addressed in later parts of the chapter.

- **Tree Shaking** excludes all parts of the compiled program that are not reachable from the program's entry points. This eliminates code from the original program or from libraries that is not used as well as auxiliary definitions that are introduced during the compilation process but not required in the final executable. (In the latter case, for example, it is not necessary to retain the code for a standalone function if all of its uses have been inlined. In our current setting, tree shaking is performed by the `shake()` function (Section 1.2), which is called frequently during optimization to identify the strongly-connected components in the current MIL program.
- **Monad Laws** are used to simplify code sequences by eliminating unnecessary uses of `return`:
  - The "right monad law" specifies that `(v <- return a; c)` has the same effect as `[a/v]c`, removing the binding for `v` and replacing all of its free occurrences in the code sequence `c` with the atom `a`. In the conventional terminology of optimizing compilers, this can be viewed as a form of either "copy propagation" (if `a` is a variable) or "constant propagation" (if `a` is a numeric constant).
  - The "left monad law" specifies that `(v <- t; return v) == t`. In more conventional terminology, we can think of this as a general

scheme for introducing tail calls, replacing a regular call to the tail `t` followed by a `return` with a direct jump to `t` instead.

The third, standard monad law (associativity) is also used indirectly in the optimization of MIL code as a way to justify the prefix inlining transformation described below.

- **Inlining** is a general technique that replaces a call to a function with the body of that function. This has the potential to increase code size, so its use should be limited. At the same time, however, inlining is also important as a transformation that eliminates the overhead that is associated with a function call, and possibly exposes new opportunities for optimization, for example, by enabling a transformation that combines code from within the inlined function call with code from the surrounding context. We distinguish between two forms of inlining for MIL:
  - **Prefix inlining** replaces a call at the beginning of a code sequence. For example, given a block definition `b(x) = (v <- t; s)`, prefix inlining might be used to rewrite code of the form `(u <- b(x); c)` as `(v <- t; u <- s; c)`. Rewrites of this kind can be understoood as applications of the third monad law, which is sometimes refered to as an "associativity" property, and can be described by an equality `(v <- t; (u <- s ; c)) == (u <- (v <- t; s); c)`; The right-hand side of this equality (which can also be obtained informally by expanding the definition of `b(x)` in `(u <- b(x); c)`), however, is not valid MIL syntax. Instead, the design of MIL essentially forces us to build in associativity so that we can rewrite any nested code sequence as a single, flattened sequence of monadic binds.
  - **Suffix inlining** replaces a call at the end of a code sequence. For example, given a block definition `b(x) = c` for some code sequence `c`, suffix inlining might be used to rewrite a code fragment of the form `(v <- t; b(v))` as `(v <- t; [v/x]c)`, where `[v/x]c` represents the result of substituting every occurrence of `x` in `c` with `v`. If we think of MIL block definitions as a kind of basic block, then this transformation corresponds to joining two sequential basic blocks by replacing an unconditional jump at the end of the first block with a copy of the code for the second.
- **Constructor Function Simplifications** Section 2.2 uses information about constructor functions to simplify code eliminating uses of newtype constructor functions and removing redundant default branches in `Match`es.
- **Simple Flow Analysis**
- **Constant Folding and Simplification**

- **Derived Blocks**

  `(v <- b(x); invoke v) == b'(x)`

  `(f <- b(x); f @ v) == b'(x, v)`

- **Liveness Analysis and Dead Code Elimination** are used to determine which variable bindings are producing values that might be referenced later in the computation, and to eliminate bindings that have no observable effects on the program's execution.

- **Common Subexpression Elimination**

  `(v <- m[x]; invoke v) == (v <- m[x]; m(x))`

lifting allocators

```
package mil;
```

2.1 General Framework

The optimizer uses a broad collection of program transformation techniques with the goal of eliminating redundant, repeated, or inefficient computations when the output program is executed. As the optimizer makes a pass over the input program, it counts the total number of transformations that have been made. This information gives us some sense of how effective the optimizer has been. If the total count is non zero, then we can conclude that some transformations have been made, possibly changing the program in ways that will enable further transformations to be made on a subsequent pass. On the other hand, if the count is zero, then the optimizer was unable to find any potential targets for improving the program, and there is no benefit in attempting to run the optimizer again.

We will record the total number of transformations that the optimizer has made in the global variable `MILProgram.count`, whose value is incremented by calling the `report()` method shown in the following code:

```
class MILProgram {
  public static int count = 0;

  public static void report(String msg) {
      debug.Log.println(msg);
      count++;
  }
}
```

Note that the `msg` argument for `report` is displayed on the debugging log so that we can track which optimizations are being applied as the optimizer runs, but it is otherwise ignored.

The entry point to the optimizer is provided by the `optimize()` method in the `MILProgram` class:

```
class MILProgram {

  /** Limit the maximum number of optimization passes that will be performed on
   *  any MIL program.  The choice is somewhat arbitrary; a higher value might
   *  allow more aggressive optimization in some cases, but might also result in
   *  larger output programs as a result of over-specialization, unrolling, or
   *  inlining.
   */
  public static final int MAX_OPTIMIZE_PASSES = 20;

  /** Run the optimizer on this program.
   */
  public void optimize() {
      shake();
      cfunSimplify();

      int totalCount = 0;
      count          = 1;
      for (int i=0; i<MAX_OPTIMIZE_PASSES && count>0; i++) {
          debug.Log.println("-------------------------");
          count = 0;
          inlining();
          debug.Log.println("Inlining pass finished, running shake.");
          shake();
          liftAllocators();  // TODO: Is this the right position for liftAllocators?
          eliminateUnusedArgs();
          flow();
          debug.Log.println("Flow pass finished, running shake.");
          shake();
          debug.Log.println("Steps performed = " + count);
          totalCount += count;
      }
      count = 0;
      collapse(); // TODO: move inside loop?
      shake();    // restore SCCs
      totalCount += count;
      debug.Log.println("TOTAL steps performed = " + totalCount);
  }

}
```

After an initial, one-time only call to `cfunSimplify()` to simplify some uses of constructor functions (described further in Section 2.2), the optimizer enters a loop that makes multiple passes over the input program, stopping if either it is unable to find any ways to transform the program (i.e., if it ends a pass with a zero value in `count`) or if it reaches some limit in the total number of iterations (set arbitrarily by the value of `MAX_OPTIMIZE_PASSES` in the code above). The latter puts an upper bound on the amount of work that the optimizer will do, and it is an attempt to deal with pathological cases that might otherwise send the optimizer in to an infinite loop. Throughout this process, the optimizer also makes frequent calls to `shake()`, described in Section 1.2, to eliminate unused definitions and to ensure that we maintain an accurate list of strongly-connected components after each set of program transformations.

The main optimization steps inside the loop are `inlining()`, `liftAllocators()`, `eliminateUnusedArgs()`, and `flow()`. As the names suggest: the `inlining()` method performs inlining transformations, such as replacing a call to a block with a copy of the code for that block; the `eliminateUnusedArgs()` method rewrites the program to eliminate unused (and hence unnecessary) `formals`

in `Blocks` and `stored` fields in `ClosureDefns`; and the `flow()` method performs simple flow-based analysis and optimizations, such as constant and copy propagation and common subexpression elimination. The `liftAllocators()` method, on the other hand, does not implement a standard optimization technique: as we will describe later, its primary purpose is to reorder the instructions in code sequences in a way that can improve the effectiveness of the subsequent flow analysis. Finally, we should also mention that each of the `inlining()` and `flow()` methods implements further optimizations that might not traditionally be captured by these single word descriptions. For example, applications of the right monad law (i.e., tail call introduction) are handled by `inlining()`, while `flow()` provides a natural place to deal with constant folding.

One of the more unusual aspects of our optimizer is the mechanisms that it uses for deriving new blocks by copying and adapting the code sequences in existing blocks. Although this has the potential to increase overall program size as a result of duplicating code, the resulting derived blocks often expose more opportunities for optimization. In addition, as calls to the original blocks are replaced by calls to the new derived blocks, it is possible that we will actually eliminate all references to the original blocks. When this happens, the original blocks become dead code that will be removed from the program as a result of running `shake()`, in which case the overall code size may not, in fact, increase. The implementation of derived blocks is described in more detail in Section 2.3 ahead of the descriptions for `inlining()` in Section 2.4 and `flow()` in Section 2.7, both of which make use of (different types of) derived blocks. The descriptions of `liftAllocators()` and `eliminateUnusedArgs()` appear in Sections 2.5 and 2.6, respectively.

2.2 Constructor Function Simplification

This section describes an initial rewrite pass over a MIL program that can simplify generated code by using information about constructor functions. Two distinct transformation techniques are used, both of which are idempotent, with the consequence that we only need to run this pass once at the start of the optimization process.

- **Eliminating newtype constructors:** We refer to a constructor function `C` as a *newtype constructor* if it has arity one and it is the only constructor for a particular type. In the syntax of Haskell, this would mean that `C` is introduced by a definition of the form `data T a1 ... an = C t` for some type `t` and type parameters `a1`, ..., `an`. In Haskell, this definition introduces some lifting; the semantics of `T a1 ... an` includes an extra bottom element that is not present in `t`. In a strict functional language, however, we treat these two types as isomorphic, with applications of `C` and pattern matching against `C` behaving as the two halves of the isomorphism. Under these circumstances, there is an opportunity to simplify

MIL code by eliminating all uses of the constructor `C`. This transformation does have a cost because it loses typing information by removing the distinction between the types `T a1 ... an` and `t`. But this is really a front-end concern (where the definition of type `T` might have been specifically introduced to to force a distinction between the two types), and eliminating constructors like this can enable further optimizations in the back-end. (We note also that none of the other optimizations described in this document relies on the elimination of newtype constructors, so this optimization could easily be disabled.)

- **Eliminating default branches:** The syntax for a `Match` allows for a default branch to be used when none of the other alternatives applies. The code generation scheme described in Section **??**, however, always includes a default branch, even if the `Match` includes a full list of alternatives that cover all possible cases. By detecting `Match`es like this where defaults can be eliminated, we can simplify the generated code, and potentially identify and eliminate more dead code.

We perform these transformations in a single pass over the abstract syntax of MIL, beginning with a loop that visits each of the definitions (in each of the strongly-connected components) in the current program.

```
/** Apply constructor function simplifications to this program.
 */
void cfunSimplify()
  case MILProgram {
      for (DefnSCCs dsccs = sccs; dsccs!=null; dsccs=dsccs.next) {
          for (Defns ds=dsccs.head.getBindings(); ds!=null; ds=ds.next) {
              ds.head.cfunSimplify();
          }
      }
  }

  case Defn abstract;
  case ClosureDefn, TopLevel { tail = tail.removeNewtypeCfun(); }
  case Block                 { code = code.cfunSimplify(); }
```

This process continues down to each `Tail` expression in the program where we replace any `DataAlloc` expression of the form `C(x)` where `C` is a newtype constructor with a simple `Return x`. In practice, many of these `Return`s are likely to be eliminated by subsequent monad law rewrites. Of course, other forms of `Tail` expression are not modified.

```
/** Eliminate a call to a newtype constructor in this Tail by replacing it
 *  with a tail that simply returns the original argument of the constructor.
 */
Tail removeNewtypeCfun()
  case Tail {  // Default case: return the original tail
      return this;
  }
  case DataAlloc {
      if (c.isNewtype()) {  // Look for use of a newtype constructor
          if (args==null || args.length!=1) {
              debug.Internal.error("newtype constructor with arity!=1");
```

```
            }
            return new Return(args[0]); // and generate a Return instead
        }
        return this;
    }
```

Of course, testing for a newtype constructor is straightforward:

```
/** Return true if this is a newtype constructor (i.e., a single argument
 *  constructor function for a type that only has one constructor).
 */
boolean isNewtype()
  case Cfun {
      return constrs.length==1 && arity==1;
      // num==1 is also required but is implied by constrs.length==1
  }
```

We also need to traverse each of the `Code` sequences in the program. The key steps required here visit each `Tail` with the `removeNewtypeCfun()` method described above, and to determine if we can remove the default branch in each `Match`s that we encounter.

```
/** Simplify uses of constructor functions in this code sequence.
 */
Code cfunSimplify()
  case Code  abstract;
  case Bind {
      t = t.removeNewtypeCfun();
      c = c.cfunSimplify();
      return this;
  }
  case Done {
      t = t.removeNewtypeCfun();
      return this;
  }
  case Match {
      // If there are no alternatives, replace this Match with a Done:
      if (alts.length==0) { // no alternatives; use default
          return (def==null) ? Code.halt : new Done(def);
      }

      // Determine which constructor numbers are covered by alts:
      boolean[] used = null;
      for (int i=0; i<alts.length; i++) {
          used = alts[i].cfunsUsed(used);
      }

      // Count to see if all constructor numbers are used:
      int count   = 0;
      int notused = 0;
      for (int i=0; i<used.length; i++) {
          if (used[i])
              count++;      // count this constructor as being used
          else
              notused = i;  // record index of an unused constructor
      }

      // If all constructor numbers have been listed, then we can eliminate
      // the default case and look for a possible newtype match:
      if (count==used.length) {
          if (count==1) {  // Look for a newtype alternative:
              Tail t = alts[0].isNewtypeAlt(a);
```

```
            if (t!=null) {
                return new Done(t);
            }
        }
        def = null;     // Eliminate the default case
    } else if (count==used.length-1 && def!=null) {
        // Promote a default to a regular alternative for better flow results:
        alts = TAlt.extendAlts(alts, notused, def); // Add new alternative
        def  = null;                                // Eliminate default
    }
    return this;
  }

class TAlt {
    /** Extend a list of (n-1) alternatives for a type that has n constructors
     *  with an extra alternative for the missing nth constructor to avoid the
     *  need for a default branch.  We assume there is at least one alternative
     *  in alts, and that the unused constructor is at position notused in the
     *  underlying array of constructors.
     */
    static TAlt[] extendAlts(TAlt[] alts, int notused, BlockCall bc) {
        int    lastIdx = alts.length;   // index of last/new alternative
        TAlt[] newAlts = new TAlt[lastIdx+1];
        for (int i=0; i<lastIdx; i++) { // copy existing alternatives
            newAlts[i] = alts[i];
        }
        Cfun c           = alts[0].c.getConstrs()[notused];
        newAlts[lastIdx] = new TAlt(c, Temp.makeTemps(c.getArity()), bc);
        MILProgram.report("Replacing default branch with match on " + c.getId());
        return newAlts;
    }
}
```

For the purposes of determining whether the default in a `Match` can be eliminated, we use the following `cfunsUsed()` method to calculate the set of constructor numbers that appear in the associated array of `TAlt` alternatives; this set is represented by the array `used` of `boolean`s in the code above. (If there are no alternatives, then we can eliminate the `Match` and replace it with a `Done` using the code from the default branch.) We assume here that the MIL code is type correct so that each of the constructors in the array of `TAlt`s will belong to the same type. The `used` array is initially `null`, but can be properly initialized to the appropriate length once we have seen the first constructor in the `TAlt` array and determined how many distinct constructors to expect in the full list. If every entry in `used` has been set by the time we have visited all of the alternatives, then we can conclude that the default branch is not needed.

```
/** Account for the constructor that is used in this alternative, setting the
 *  corresponding flag in the argument array, which has one position for each
 *  constructor in the underlying type.
 */
boolean[] cfunsUsed(boolean[] used)
  case TAlt {
      // Allocate a flag array based on the constrs array for c:
      if (used==null) {
   used = new boolean[c.getConstrs().length];
      }
      // Flag use of this constructor as having been mentioned:
      int num = c.getNum();    // flag use of this constructor
      if (num>=0 && num<used.length) {
   if (used[num]) {
       debug.Internal.error("multiple alternatives for " + c);
```

```
    }
          used[num] = true;
       } else {
     debug.Internal.error("cfun index out of range");
       }
       return used;
   }
```

If the list of alternatives in a `Match` includes an alternative for each of the constructors in a given type, then the default branch can be eliminated. As a special case, if we are dealing with a `TAlt` that uses a newtype constructor, then we can go a step further, eliminating the `Match` altogether and substituting a `Done` in its place. The following code tests for this special case, returning a `BlockCall` that uses the given atom (the discriminant from the `Match`) in place of the variable that was used in the pattern when it succeeds, or otherwise returning `null`.

```
/** Determine if this alternative is for a newtype constructor, in which case return
 *  the simple block call that can be used to replace it.
 */
BlockCall isNewtypeAlt(Atom a)
  case TAlt {
      return (c.isNewtype())
                ? bc.forceApplyBlockCall(new AtomSubst(args[0], a, null))
                : null;
  }
```

2.3 Derived Blocks

There are a number of situations in which we might want to derive a new block from an existing block of code in the hope of producing more efficient code. Suppose, for example, that we encounter code of the following form in a sequence where `v` is not live after the `invoke`.

```
v <- b(x,y)
t <- invoke v
c
```

A likely scenario here is that the code for `b(x,y)` allocates a thunk that is returned as the result of the call, immediately invoked, and then discarded. In situations like this, it might be better to arrange for the allocated thunk to be invoked immediately before returning from the block, instead of immediately after it. This might then allow us to replace the invocation with a direct call and perhaps remove the thunk allocation as dead code. For the example here, this would mean replacing the original code sequence with the following:

```
t <- b'(x,y)
c
```

where `b'` is a variant of `b` that invokes its result before returning, instead of returning a thunk. For example, if `b` is defined by a code sequence of the form:

```
b(x,y) =
  v1 <- t1
  v2 <- t2
  ...
  vn <- tn
  t
```

then we can generate the following definition for the derived block `b'`:

```
b'(x,y) =
  v1 <- t1
  v2 <- t2
  ...
  vn <- tn
  v  <- t
  invoke v
```

The code transformation used here is quite straigtforward; we have just copied the initial sequence of binds directly from the original code in `b`, but then instead of just returning `t`, we capture the result (which should be a thunk) with a fresh variable `v` and then immediately invoke it using `invoke v`.

This is potentially useful because including the `invoke` step explicitly in the code for `b'` may produce an opportunity to eliminate the construction of a thunk (for example, if `t` is a thunk allocator of the form `m[...]`). By comparison, the original call to `b`, followed by a separate `invoke`, will necessarily force the use, and often the allocation of a monadic thunk.

A similar transformation is useful in cases where the result of a block call is a closure that is immediately entered with a specific argument value and then never used again, as in the following (with the assumption that `f` is not used in `c`):

```
f <- b(x,y)
u <- f @ a
c
```

In this case, we might want to introduce a new block `b'` that is like `b` except that it adds an extra parameter for the argument `a`, and enters the closure that would be built by the code in `b` before returning. With this transformation, the call of `b` in the original code sequence would be replaced by:

```
u <- b'(x,y,a)
c
```

And, assuming a definition for `b` of the same form shown previously, the new block `b'` would be introduced by a definition of the following form with fresh variables `f` and `a`:

```
b'(x,y,a) =
  v1 <- t1
  v2 <- t2
  ...
  vn <- tn
  f  <- t
  f @ a
```

Translations like this can be made for any block, even if the associated code sequence does not end in a simple tail (i.e., if it is a `Match`), although this may also require the definition of other new blocks with an extra argument. For example, a definition for `b` of the form:

```
b(x,y) =
  v1 <- t1
  v2 <- t2
  ...
  vn <- tn
  case v of
    p1 -> b1(...)
    ...
    pm -> bm(...)
```

can be used to create a corresponding definition for `b'`:

```
b'(x,y,a) =
  v1 <- t1
  v2 <- t2
  ...
  vn <- tn
  case v of
    p1 -> b1'(..., a)
    ...
    pm -> bm'(..., a)
```

In this code, `b1'`, ..., `bm'` are similarly derived blocks corresponding to the original blocks `b1`, ..., `bm`. This is a risky translation to apply in general because it has the potential to duplicate a lot of code. However, if `b` has a small body, or if it is only ever called when a known argument is available (so that all calls to `b` are replaced by calls to `b'` and the original definition for `b` becomes

dead code), then the risks seem less severe. Moreover, if we assume that the code we are optimizing is type correct, then this particular transformation will only ever be applied to blocks that return values of function type, limiting the potential for code duplication to a fraction of the overall program.

Of course, it is possible for a particular program to contain multiple instances of patterns like the ones described above, all of which start with the same block, `b`. To ensure that we do not end up deriving multiple copies of the same new block, we add a list in each 'Block' structure to record all of the new blocks that 'derived' from it. When an opportunity to use a derived block occurs, we consult the list first to check that the required new block has not already been constructed.

```
macro List(Block)    // Introduces class Blocks, representing lists of Block values

class Block {
  /** Stores the list of blocks that have been derived from this block.
   */
  private Blocks derived = null;
}
```

In addition, we define a simple `isBlockCall()` method as a way to distinguish `BlockCall` objects from other `Tail` values. For example, by using this method on the representation for `t` in a code sequence of the form `(v <- t; c)`, we can determine if `t` is a block call, and hence make a first step in determining whether this code might provide an opportunity for using a derived block.

```
/** Test to determine if this Tail or Code value is a BlockCall.
 */
public BlockCall isBlockCall()
  case Tail       { return null; }
  case BlockCall  { return this; }
```

In the following subsections, we describe four specific techniques for generated derived blocks, including the two examples described previously that add trailing 'invoke' and enter operations ('f @ a') at the end of the derived block. In each case, we introduce a subclass of 'Block' to represent the new form of derived block; we define a function that can construct a new derived block of the appropriate form on demand, checking the 'derived' list first to see if the required block has already been constructed; we add methods to construct the code for the new block; and we define predicates for detecting situations in which each form of derived block could be useful.

2.3.1 Adding a Trailing Invoke

Our first example of a derived block corresponds to the first example described previously in which we transform code sequences like `(w <- b(x..); invoke w)` into calls of the form `b'(x..)` where `b'` is a new block that has been derived from the code for `b` by adding a trailing `invoke`. Although there is a risk of code

duplication, this transformation might also enable later optimization steps: to replace the `invoke` operation with a direct call; to eliminate the need to allocate a thunk; and perhaps to identify and delete the original definition of the block `b` if its uses can all be replaced by corresponding uses of `b'`.

In concrete terms, we implement these ideas by defining a `deriveWithInvoke()` method such that `b.deriveWithInvoke()` will either create a new version of `b`, or else return a previously specialized version from `b`'s `derived` list. To distinguish blocks that have been obtained in this way from other forms of derived block, we will represent them using objects of type `BlockWithInvoke`, which is defined as a new subclass of `Block`.

```
/** Represents a block that was derived by adding a trailing invoke to
 *  its code sequence.
 */
public class BlockWithInvoke extends Block(private Code code)

class Block {
  /** Derive a new version of this block using a code sequence that invokes
   *  the final result before it returns instead of returning that value
   *  (presumably a thunk) to calling code where it is immediately invoked
   *  and then discarded.
   */
  public Block deriveWithInvoke() {
      // Look to see if we have already derived a suitable version of this block:
      for (Blocks bs = derived; bs!=null; bs=bs.next) {
          if (bs.head instanceof BlockWithInvoke) {
              return bs.head;
          }
      }

      // Generate a fresh block; we have to make sure that the new block is
      // added to the derived list before we begin generating code to ensure
      // that we do not end up with multiple (or potentially, infinitely
      // many copies of the new block).
      Block b   = new BlockWithInvoke(null);
      derived   = new Blocks(b, derived);
      b.formals = this.formals;
      b.code    = code.deriveWithInvoke();
      return b;
  }
}
```

We also define a `deriveWithInvoke()` method on `Code` values to describe the construction of the required code sequences for blocks that are derived in this way. In essence, `code.deriveWithInvoke()` just recurses through and makes a copy of the steps in the original `code` sequence before appending the extra `Invoke` step when it reaches `Done` at the end of the original. For code sequences that end with a `Match` rather than a `Done`, we perform recursive `deriveWithInvoke()` calls for each of the possible continuation blocks to ensure that the required trailing `invoke` will always be executed (assuming that the code in the original block would have ever returned). In theory, this might mean that we end up deriving multiple new blocks as a result of the initial `deriveWithInvoke()` call. Thanks to the caching effect of the `derived` list, however, we will create at most one new block for each original.

```
/** Modify this code sequence to add a trailing invoke operation.
```

```
 */
Code deriveWithInvoke()
  case Code abstract;
  case Done {
      Var v = new Temp();
      return new Bind(v, t, new Done(new Invoke(v)));
  }
  case Bind {
      return new Bind(v, t, c.deriveWithInvoke());
  }
  case Match {
      TAlt[] nalts = new TAlt[alts.length];
      for (int i=0; i<alts.length; i++) {
          nalts[i] = alts[i].deriveWithInvoke();
      }
      BlockCall d = (def==null) ? null : def.deriveWithInvoke();
      return new Match(a, nalts, d);
  }

/** Generate a new version of this alternative that branches to a
 *  derived block with a trailing invoke.
 */
public TAlt deriveWithInvoke()
  case TAlt {
      return new TAlt(c, args, bc.deriveWithInvoke());
  }

/** Generate a new version of this block call that branches to a
 *  block with a trailing invoke.
 */
public BlockCall deriveWithInvoke()
  case BlockCall {
      BlockCall bc = new BlockCall(b.deriveWithInvoke());
      bc.withArgs(args);
      return bc;
  }
```

Now that we have the infrastructure for creating new blocks, we define the following `invokes()` methods to identify code sequences where they can be used, and to perform the appropriate program transformations. One situation where this occurs, as described above, is in code sequences of the form `(w <- bc; invoke w)` where `bc` is a block call, which can then be rewritten as a single derived block call of the form `bc'`. However, it is not necessary to require that 'invoke w' appears at the very end of a code sequence because we can also apply the same basic idea by replacing `invoke w` with `bc'` in a code sequence of the form `(w <- bc; v <- invoke w; c)` to obtain `(w <- bc'; c)`, provided that there are no references to `w` in `c`. These two cases correspond to implementations of `invokes(w, bc)` for the `Done` and `Bind` classes, respectively:

```
/** Given an expression of the form (w <- b(..); c), attempt to construct an
 *  equivalent code sequence that instead invokes a block whose code includes
 *  a trailing invoke.
 */
public Code invokes(Var w, BlockCall bc)
  case Code {
      return null;
  }
  case Done {
      return t.invokes(w) ? new Done(bc.deriveWithInvoke()) : null;
  }
  case Bind {
```

```
      return (t.invokes(w) && !c.contains(w))
                ? new Bind(v, bc.deriveWithInvoke(), c) : null;
  }

/** Test to determine if this Tail is an expression of the form (invoke v).
 */
public boolean invokes(Var v)
  case Tail   { return false; }
  case Invoke { return v==a;  }
```

2.3.2 Adding a Trailing Enter

The same basic pattern that we used in the previous section can be adapted to describe the construction of derived blocks that add a trailing enter; which was the second example that we used to motivate the use of derived blocks at the start of Section 2.3. In this case, the goal is to recognize code sequences of the form `(f <- bc; f @ a)` where the block call `bc` returns a function value, `f`, that is immediately applied to some known argument, `a`, and then discarded. If `bc'` is a derived version of the block call `bc` that accepts `a` as an extra argument, and ends by using `f @ a` instead of returning `f` as its result, then we can use `bc'` in place of the original code sequence. In a similar way, if `f` and `v` are distinct and `f` does not appear in `c`, then we can use the derived block call `bc'` to rewrite `(f <- bc; v <- f @ a; c)` as `(v <- bc'; c)`. In some situations, moving the code to enter the closure from the caller to the callee like this will enable later stages of the optimization process to replace the closure entry with a direct call, and even to avoid constructing the closure in the first place.

We represent derived blocks produced in this way as objects of `BlockWithEnter`, which is a new subclass of `Block`, and we provide a `deriveWithEnter()` method to handle the process of checking for and returning a previously derived block if possible, but otherwise generating a new block of the appropriate form (including, in particular, an additional formal parameter to pass in the extra argument, `a`):

```
/** Represents a block that was derived by adding a trailing enter to
 *  its code sequence.
 */
public class BlockWithEnter extends Block(private Code code)

class Block {
  /** Derive a new version of this block using a code sequence that applies
   *  its final result to a specifed argument value instead of returning
   *  that value (presumably a closure) to the calling code where it is
   *  immediately entered and then discarded.
   */
  public Block deriveWithEnter() {
      // Look to see if we have already derived a suitable version of this block:
      for (Blocks bs = derived; bs!=null; bs=bs.next) {
          if (bs.head instanceof BlockWithEnter) {
              return bs.head;
          }
      }

      // Generate a fresh block; we have to make sure that the new block is
      // added to the derived list before we begin generating code to ensure
```

```
      // that we do not end up with multiple (or potentially, infinitely
      // many copies of the new block).
      Block b   = new BlockWithEnter(null);
      derived   = new Blocks(b, derived);
      Temp arg  = new Temp();
      int l     = formals.length; // extend formals with arg
      Var[] nfs = new Var[l+1];
      for (int i=0; i<l; i++) {
          nfs[i] = formals[i];
      }
      nfs[l]    = arg;
      b.formals = nfs;
      b.code    = code.deriveWithEnter(arg);
      return b;
  }
}
```

The process of generating code for a block with a trailing enter also follows much the same pattern that we saw in the previous section for blocks with a trailing invoke, but adding an extra parameter to specify the name of the final argument.

```
/** Modify this code sequence to add a trailing enter operation that
 *  applies the value that would have been returned by the code in
 *  the original block to the specified argument parameter.
 */
Code deriveWithEnter(Atom arg)
  case Code abstract;
  case Done {
      Var f = new Temp();
      return new Bind(f, t, new Done(new Enter(f,arg)));
  }
  case Bind {
      return new Bind(v, t, c.deriveWithEnter(arg));
  }
  case Match {
      TAlt[] nalts = new TAlt[alts.length];
      for (int i=0; i<alts.length; i++) {
          nalts[i] = alts[i].deriveWithEnter(arg);
      }
      BlockCall d = (def==null) ? null : def.deriveWithEnter(arg);
      return new Match(a, nalts, d);
  }

/** Generate a new version of this alternative that branches to a
 *  derived block with a trailing enter.
 */
public TAlt deriveWithEnter(Atom arg)
  case TAlt {
      return new TAlt(c, args, bc.deriveWithEnter(arg));
  }

/** Generate a new version of this block call by passing the
 *  specified argument to a derived block with a trailing enter.
 */
public BlockCall deriveWithEnter(Atom arg)
  case BlockCall {
      BlockCall bc = new BlockCall(b.deriveWithEnter());
      int l        = args.length;
      Atom[] nargs = new Atom[l+1]; // extend args with arg
      for (int i=0; i<l; i++) {
          nargs[i] = args[i];
      }
      nargs[l] = arg;
      bc.withArgs(nargs);
```

```
        return bc;
    }
```

All that remains before we can start using `deriveWithEnter()` is the following code for identifying the situations in which this form of derived block can be used, and for generated the appropriately transformed code sequences:

```
/** Given an expression of the form (w <- b(..); c), attempt to construct an
 *  equivalent code sequence that instead calls a block whose code includes a
 *  trailing enter.
 */
public Code enters(Var w, BlockCall bc)
  case Code { return null; }
  case Done {
      Atom arg = t.enters(w);
      return (arg!=null) ? new Done(bc.deriveWithEnter(arg)) : null;
  }
  case Bind {
      Atom arg = t.enters(w);
      return (arg!=null && !c.contains(w))
         ? new Bind(v, bc.deriveWithEnter(arg), c) : null;
  }

/** Test to determine if this Tail is an expression of the form (v @ a)
 *  for some given v, and any a (except v), returning the argument a as
 *  a result.
 */
public Atom enters(Var v)
  case Tail  { return null; }
  case Enter { return (f==v && a!=v) ? a : null;  }
```

2.3.3 Adding a Continuation

Continuing the pattern from the previous two sections, it is natural to consider the possibility of optimizing code sequences of the form `(a <- bc; f @ a)` where the block call `bc` produces a result `a` that is immediately passed as an argument to some fixed function `f` and then discarded. In this situation, we can derive a new block and a corresponding block call `bc'` that takes the function `f` as an extra argument, effectively serving as a *continuation*, and then passes the original result `a` to the parameter `f` before it returns. As such, we will refer to this transformation as deriving a block with an added continuation, and we will implement it using the `deriveWithCont()` methods that are defined below.

Although this seems to be essentially a dual of the `deriveWithEnter()` scheme that was described in the previous section, we have not seen many occurrences of this particular pattern in practice. Moreover, it is not clear if the benefits outweight the costs in this case; the transformation essentially forces the allocation of a closure for `f` so that it can be passed as an argument, which might not have been necessary with the original code sequence. For these reasons, our optimizer does not currently attempt to apply these transformations to user code. (In concrete terms, this means that it does not attempt to perform a rewrite in situations where the `appliesTo()` method described below indicate that a rewrite would be possible.

Nevertheless, we still include the code for `deriveWithCont()` in this section because we have found good uses for derived blocks of this form in situations where a program transformation is most easily described by adding an artificial continuation. For example, in a code sequence `(v <- bc; case v of alts)`, where the result of the block call `bc` is immediately inspected by a `case` construct, we may wish to push the `case` construct into the individual alternatives in `alts`. This, for example, might allow us to avoid allocating heap memory for the value that is returned by `bc` in the original code if it leads to code for `bc'` of the form `(v <- C(...); case v of alts)` where the allocation step can be paired with a specific alternative and then eliminated as dead code. Implementing this kind of transformation directly has proved to be messy. However, it is relatively easy to express this if we define a new closure `k` such that `k{...} v = case v of alts` and then replace the original code sequence with `(f <- k{...}; bc')` where `bc'` is a derived (`deriveWithCont()`) block call with the continuation `f` as an extra parameter. Although the immediate results of this particular transformation are not particularly good—in particular, this code requires an extra closure allocation—some of the other optimization techniques that we have implemented (for example, using the `deriveWithKnownCons()` method that is described in the next section) are directly applicable to code sequences of this form. As such, we can expect to see inprovements in the code after additional iterations of the optimizer.

The implementation of `deriveWithCont()` follows exactly the same pattern as the previous two examples, this time using a new subclass of `Block` called `BlockWithCont` to distinguish blocks that are derived in this way.

```
/** Represents a block that was derived by adding a trailing invocation
 *  of a new continuation argument to its code sequence.
 */
public class BlockWithCont extends Block(private Code code)

class Block {
  /** Derive a new version of this block using a code sequence that passes
   *  its final result to a specifed continuation function instead of returning
   *  that value to the calling code.
   */
  public Block deriveWithCont() {
      // Look to see if we have already derived a suitable version of this block:
      for (Blocks bs = derived; bs!=null; bs=bs.next) {
          if (bs.head instanceof BlockWithCont) {
              return bs.head;
          }
      }

      // Generate a fresh block; we have to make sure that the new block is
      // added to the derived list before we begin generating code to ensure
      // that we do not end up with multiple (or potentially, infinitely
      // many copies of the new block).
      Block b   = new BlockWithCont(null);
      derived   = new Blocks(b, derived);
      Temp arg  = new Temp();     // represents continuation
      int l     = formals.length; // extend formals with arg
      Var[] nfs = new Var[l+1];
      for (int i=0; i<l; i++) {
          nfs[i] = formals[i];
      }
```

```
      nfs[l]    = arg;
      b.formals = nfs;
      b.code    = code.deriveWithCont(arg);

      return b;
  }
}
```

The task of generating a derived code sequence for `deriveWithCont()` also follows a now familiar pattern. In comparison with the `deriveWithEnter()` methods that were described in the previous section, the only real difference here is a small change in the details of how `Done` nodes are handled.

```
/** Modify this code sequence to add a trailing enter operation that
 *  passes the value that would have been returned by the code in
 *  the original block to the specified continuation parameter.
 */
Code deriveWithCont(Atom cont)
  case Code abstract;
  case Done {
      Var v = new Temp();
      return new Bind(v, t, new Done(new Enter(cont,v)));
  }
  case Bind {
      return new Bind(v, t, c.deriveWithCont(cont));
  }
  case Match {
      TAlt[] nalts = new TAlt[alts.length];
      for (int i=0; i<alts.length; i++) {
          nalts[i] = alts[i].deriveWithCont(cont);
      }
      BlockCall d = (def==null) ? null : def.deriveWithCont(cont);
      return new Match(a, nalts, d);
  }

/** Generate a new version of this alternative that branches to a
 *  derived block with a trailing invocation of a continuation.
 */
public TAlt deriveWithCont(Atom cont)
  case TAlt {
      return new TAlt(c, args, bc.deriveWithCont(cont));
  }

/** Generate a new version of this block call by passing the specified
 *  continuation argument to a derived block with a trailing continuation
 *  invocation.
 */
public BlockCall deriveWithCont(Atom cont)
  case BlockCall {
      BlockCall bc = new BlockCall(b.deriveWithCont());
      int l        = args.length;
      Atom[] nargs = new Atom[l+1]; // extend args with arg
      for (int i=0; i<l; i++) {
          nargs[i] = args[i];
      }
      nargs[l] = cont;
      bc.withArgs(nargs);
      return bc;
  }
```

We can identify opportunities for using blocks produced by `deriveWithCont()` by looking for code sequences either of the form `(w <- bc; f @ w)` or else of the form `(w <- bc; v <- f @ w; c)` where `w` and `f` are distinct and `a` does

not appear in c. These tests, and the corresponding transformations in each case, are implemented by the following methods.

```
/** Given an expression of the form (w <- b(..); c), attempt to construct an
 *  equivalent code sequence that instead calls a block whose code will attempt
 *  to call a continuation on the result instead of returning it directly.
 */
public Code appliesTo(Var w, BlockCall bc)
  case Code { return null; }
  case Done {
      Atom cont = t.appliesTo(w);
      // TODO: replace this with the commented section below to make this live ...
      return (cont!=null) ? this /* new Done(bc.deriveWithCont(cont)) */ : null;
  }
  case Bind {
      Atom cont = t.appliesTo(w);
      return (cont!=null && !c.contains(w))
          ? new Bind(v, bc.deriveWithCont(cont), c) : null;
  }

/** Test to determine if this Tail is an expression of the form (f @ w)
 *  for some given w, and any f (except w), returning the function, f,
 *  as a result.
 */
public Atom appliesTo(Var w)
  case Tail  { return null; }
  case Enter { return (a==w && a!=f) ? f : null;  }
```

As mentioned previously, the optimizer does not currently attempt to make use of derived blocks in situations where `appliesTo()` identifies a possible transformation because of concerns that it may lead to reduced code quality. But the blocks produced by `deriveWithCont()` are still useful, however, in other settings. The code below expands on the previously mentioned example by showing how `derivedWithCont()` blocks can be used to rewrite code sequences of the form `(v <- bc; case v of alts)` by introducing a known continuation that is represented by a closure `k{...}` that is defined by `k{...} v = case v of alts`. (The arguments of the closure, represented by `...` here, are computed using the results of the liveness analysis that is defined later in Section **??**). Given that definition of `k`, we can then rewrite the original code sequence as `(f <- k{...}; bc')` where `f` is a fresh variable that is passed as an extra argument to the block call `bc` that was derived from `bc'`. With the following method definition, a call of the form `c.casesOn(v,bc)` will: first, determine whether the code sequence `c` is of the form `(case v of alts)` where `v` is the result of some preceding block call `bc`; and second, if so, generate a suitable definition for the new closure and return the transformed code sequence.

```
/** Test to determine if this code is an expression of the form case v of alts where v
 *  is the result of a preceding block call.  If so, return a transformed version of
 *  the code that makes use of a newly defined, known continuation that can subsequently
 *  be pushed in to the individual branches of the case for further optimization.
 */
public Code casesOn(Var v, BlockCall bc)
  case Code  { return null; }
  case Match {
      if (a==v && bc.contCand()) {
          // Build a block:  b(...) = case v of ...
          Code  match   = copy();               // make a copy of this Match
```

```
            Vars  vs      = match.liveness();      // find the free variables
            Block b       = new Block(match);
            Var[] formals = Vars.toArray(vs);      // set the formal parameters
            b.setFormals(formals);

            // Build a closure definition: k{...} v = b(...)
            Tail t        = new BlockCall(b).withArgs(formals);
            ClosureDefn k = new ClosureDefn(v, t); // define a new closure
            Var[] stored  = Vars.toArray(Vars.remove(v, vs));
            k.setStored(stored);                   // do not store v in the closure

            // Construct a continuation for the derived block:
            Tail cont = new ClosAlloc(k).withArgs(stored);
            Var  w    = new Temp();

            // Replace original code with call to a new derived block:
            return new Bind(w, cont, new Done(bc.deriveWithCont(w)));
        }
        return null;
    }
```

In making this transformation and introducing (rather than eliminating) a closure allocation, we are relying on the assumption that a subsequent use of the `deriveWithKnownCons()` method, which is described in the next section, will produce a further derived version of the code for `bc'` that is specialized to the continuation `k{...}`. If that happens, then it should be possible to eliminate the closure construction once it has been pushed into each of the alternatives. For this to work well, however, we will limit the use of this transformation to cases where the definition of the block referenced by `bc` satisfies a syntactic check described by the `contCand()` method described below. This particular function will only return true if the block that is being called, as well as all the blocks that it depends on either directly or indirectly, are in nonrecursive, strongly-connected components, which already covers a good range of the examples that we see in practice, although there is certainly room to refine and improve this test in the future.

```
/** Heuristic to determine if this block is a good candidate for the casesOn().
 *  TODO: investigate better functions for finding candidates!
 */
boolean contCand()
  case BlockCall { return b.contCand(); }
  case Block     { return !(this instanceof BlockWithKnownCons) && getScc().contCand(); }
  case DefnSCC {
      if (isRecursive()) {
          return false;
      }
      for (DefnSCCs ds = getDependsOn(); ds!=null; ds=ds.next) {
          if (!ds.head.contCand()) {
              return false;
          }
      }
      return true;
  }
```

2.3.4 Specializing a Block on Known Constructors

```
public class BlockWithKnownCons(private Allocator[] allocs)
```

```
  extends Block(private Code code)

// Invariant: allocs is not null.

class Block {
  public Block deriveWithKnownCons(Allocator[] allocs) {

    // Do not create a specialized version of a simple block (i.e., a block
    // that contains only a single Done):
//    if (code instanceof Done) {
//System.out.println("Will not specialize this block: code is a single tail");
//      return null;
//  }
    if (this instanceof BlockWithKnownCons) {
      return null;
    }


    // Look to see if we have already derived a suitable version of this block:
    for (Blocks bs = derived; bs!=null; bs=bs.next) {
      if (bs.head.hasKnownCons(allocs)) {
        return bs.head;
      }
    }

    // Generate a fresh block; unlike the cases for trailing Enter and
    // Invoke, we're only going to create one block here whose code is
    // the same as the original block except that it adds a group of one
    // or more initializers.  So we'll initialize the block as follows
    // and add the initializers later.
    Block b = new BlockWithKnownCons(code.copy(), allocs);
    derived = new Blocks(b, derived);

    // Create lists of arguments for individual allocators
    Var[][] fss = new Var[allocs.length][];
    int len     = 0;
    for (int i=0; i<allocs.length; i++) {
      if (allocs[i]==null) {
        len++;
      } else {
        Top top = allocs[i].getTop();
        fss[i]  = Temp.makeTemps(allocs[i].getArity());
        len    += fss[i].length;
      }
    }

    // Generate list of formals for this block:
    Var[] nfs  = new Var[len];
    int   pos  = 0;
    for (int i=0; i<allocs.length; i++) {
      if (fss[i]==null) {
        nfs[pos++] = this.formals[i];
        len++;
      } else {
        int l = fss[i].length;
        for (int j=0; j<l; j++) {
          nfs[pos++] = fss[i][j];
        }
      }
    }
    b.formals = nfs;

    // Create code for this block by prepending some initializers:
    for (int i=0; i<allocs.length; i++) {
      if (fss[i]!=null) {
        Top top = allocs[i].getTop();
        Tail t  = (top!=null /*&& fss[i].length==0 */)
                    ? new Return(top)
```

```
                : allocs[i].callDup().withArgs(fss[i]);
        b.code = new Bind(this.formals[i], t, b.code);
      }
    }

    return b;
  }
}

Code copy()
  case Code  abstract;
  case Done  { return new Done(t); }
  case Bind  { return new Bind(v, t, c.copy()); }
  case Match { return new Match(a, TAlt.copy(alts), def); }

class TAlt {
  static TAlt[] copy(TAlt[] alts) {
    if (alts==null) {
      return null;
    } else {
      TAlt[] copied = new TAlt[alts.length];
      for (int i=0; i<alts.length; i++) {
        copied[i] = alts[i].copy();
      }
      return copied;
    }
  }
}

TAlt copy()
  case TAlt { return new TAlt(c, args, bc); }
```

Checking to see if two arrays of allocator expressions are consistent is not difficult, but requires a loop that iterates across both arrays looking either for corresponding pairs of nulls or else for corresponding allocators that have the same outer constructor.

```
boolean hasKnownCons(Allocator[] allocs)
  case Block { return false; }
  case BlockWithKnownCons {
    for (int i=0; i<allocs.length; i++) {
      // Compare allocs[i] and this.allocs[i].  Both must be null, or
      // they must have the same constructor to continue to the next i.
      if (allocs[i]==null) {
        if (this.allocs[i]!=null) {
          return false;
        }
      } else if (this.allocs[i]==null || !allocs[i].sameFormAlloc(this.allocs[i])) {
        return false;
      }
    }
    return true;
  }

boolean sameFormAlloc(Allocator a)
  case Allocator abstract;
  case DataAlloc { return a.isDataAlloc(c); }
  case ClosAlloc { return a.isClosAlloc(k); }
  case CompAlloc { return a.isCompAlloc(m); }

boolean isDataAlloc(Cfun c)
  case Allocator { return false; }
  case DataAlloc { return c==this.c; }
```

```
boolean isClosAlloc(ClosureDefn k)
  case Allocator { return false; }
  case ClosAlloc { return k==this.k; }

boolean isCompAlloc(Block m)
  case Allocator { return false; }
  case CompAlloc { return m==this.m; }
```

Modify a code sequence to apply the original return value to a given argument instead of returning it as a closure.

```
public BlockCall deriveWithKnownCons(Allocator[] allocs)
  case BlockCall {
    if (allocs.length!=args.length) {
      debug.Internal.error("argument list length mismatch in deriveWithKnownCons");
    }
    Block nb = b.deriveWithKnownCons(allocs);
    if (nb==null) {
      return null;
    } else {
      BlockCall bc = new BlockCall(nb);
      bc.withArgs(specializedArgs(allocs));
      return bc;
    }
  }

CompAlloc deriveWithKnownCons(Allocator[] allocs)
  case CompAlloc {
    Block nm = m.deriveWithKnownCons(allocs);
    if (nm==null) {
      return null;
    } else {
      CompAlloc ca = new CompAlloc(nm);
      ca.withArgs(specializedArgs(allocs));
      return ca;
    }
  }

class Call {
  Atom[] specializedArgs(Allocator[] allocs) {
      // Compute the number of actual arguments that are needed:
      int len = 0;
      for (int i=0; i<allocs.length; i++) {
        len += (allocs[i]==null ? 1 : allocs[i].getArity());
      }

      // Fill in the actual arguments:
      Atom[]    nargs = new Atom[len];
      int       pos   = 0;
      for (int i=0; i<args.length; i++) {
        if (allocs[i]==null) {
          nargs[pos++] = args[i];
        } else {
          pos = allocs[i].collectArgs(nargs, pos);
        }
      }

      // Return specialized list of arguments:
      return nargs;
  }
}

class Allocator {
  int collectArgs(Atom[] result, int pos) {
    for (int j=0; j<args.length; j++) {
      result[pos++] = args[j];
```

```
    }
    return pos;
  }
}
```

2.4 Inlining

Inlining is a standard compiler optimization that aims to remove runtime overhead by replacing individual calls to a function with a (suitably instantiated) copy of the associated function body. As a side-effect, inlining can also expose new opportunities for optimization. To illustrate these features, suppose that we are working with a program that contains the following block definitions:

```
b1(x,y) = v <- add((x,1)); add((v,y))
b2(x,y) = w <- add((y,1)); mul((w,x))
```

Calling a function typically requires multiple steps to save register values and create an appropriate stack frame on entry, and then dismantle the frame and restore the original register values when the function returns. Clearly, we would prefer to avoid these overheads, particularly for calls to simple functions like `b1` and `b2` in the above—each of which only requires two simple arithmetic operations. A simple inlining transformation uses the definition of `b1` and `b2` to transform a code sequence of the form:

```
t <- b1(p,q); b2(t,p)
```

into the equivalent:

```
v <- add((p,1)); t <- add((v,q)); w <- add((p,1)); mul((w,t))
```

Although this second code sequence is longer, it only uses primitive calls—which can be translated directly in to individual machine instructions—and avoids the need for more general function calls. In addition, once the definitions for `b1` and `b2` have been inlined, we can see that there are two separate steps that compute the value of `add((p,1))`. Subsequent uses of common subexpression elimination (see Section **??**) and copy propagation (Section **??**) can then further simplify this code sequence to:

```
v <- add((p,1)); t <- add((v,q); mul((v,t))
```

If the code sequence that we have just transformed was the only place in the whole program where `b1` and `b2` were called, then the optimizations described here will have eliminated the only references to those blocks, which will then be dropped from the program as dead code after the next call to `shake()`, resulting in a program that not only runs faster, but is also smaller.

On the other hand, an aggressive inlining strategy that inlines every function call could result in a signficantly larger program, duplicating the code from individual function bodies, and requiring special care to avoid unrolling recursive definitions to arbitrary depths (and sizes). Increasing program size can also increase the cost of further analysis, optimization, and other key steps in the compilation process, such as register allocation and instruction selection. For these reasons, a good inliner must strike a careful balance, inlining definitions to remove overhead when possible, but also trying to avoid a significant increase in program size.

```
class MILProgram {
  /** Run an inlining pass over this program, assuming a preceding call to
   *  shake() to compute call graph information.  Starts by performing a
   *  "return analysis" (to detect blocks that are guaranteed not to return);
   *  uses the results to perform some initial cleanup; and then performs
   *  simple loop detection to identify code that loops with no observable
   *  effect and would cause the inliner to enter an infinite loop.  With
   *  those preliminaries out of the way, we then invoke the main inliner!
   */
  public void inlining() {
    for (DefnSCCs dsccs = sccs; dsccs!=null; dsccs=dsccs.next) {
      Defns defns = dsccs.head.getBindings();

      // Initialize return analysis information for each definition in this scc:
      for (Defns ds=defns; ds!=null; ds=ds.next) {
          ds.head.resetDoesntReturn();
      }

      // Compute return analysis results for each item in this scc:
      boolean changed = true;
      do {
          changed = false;
          for (Defns ds=defns; ds!=null; ds=ds.next) {
               changed |= ds.head.returnAnalysis();
          }
      } while (changed);

      // Use results of return analysis to clean up code:
      for (Defns ds=defns; ds!=null; ds=ds.next) {
          ds.head.cleanup();
      }

      // Run loop detection on this scc, rewriting some block definitions
      // that would otherwise send the inliner in to an infinite loop:
      for (Defns ds=defns; ds!=null; ds=ds.next) {
          ds.head.detectLoops(null);
      }

      // Perform inlining on the definitions inside this scc:
      for (Defns ds=defns; ds!=null; ds=ds.next) {
          ds.head.inlining();
      }
    }
  }
}
```

2.4.1 Return Analysis

In this section, we describe a "return analysis" whose primary purpose is to identify blocks that "do not return". The

```
class Block {
  /** Flag to identify blocks that "do not return".  In other words, if the
   *  value of this flag for a given block b is true, then we can be sure that
   *  (x <- b(args); c) == b(args) for any appropriate set of arguments args
   *  and any valid code sequence c.  There are two situations that can cause
   *  a block to "not return".  The first is when the block enters an infinite
   *  loop; such blocks may still be productive (such as the block defined by
   *  b(x) = (_ <- print((1)); b(x))), so we cannot assume that they will be
   *  eliminated by eliminateLoops().  The second is when the block's code
   *  sequence makes a call to a primitive call that does not return.
   */
  private boolean doesntReturn = false;

  /** Return flag, computed by previous dataflow analysis, to indicate
   *  if this block does not return.
   */
  boolean doesntReturn() {
      return doesntReturn;
  }
}

/** Reset the doesntReturn flag, if there is one, for this definition
 *  ahead of a returnAnalysis().  For this analysis, we use true as
 *  the initial value, reducing it to false if we find a path that
 *  allows a block's code to return.
 */
void resetDoesntReturn()
  case Defn  { /* Nothing to do in this case */ }
  case Block { this.doesntReturn = true; }

class MILProgram {
  public void returnAnalysis() {
      // Calculate doesntReturn information for every Block:
      for (DefnSCCs dsccs = sccs; dsccs!=null; dsccs=dsccs.next) {
           // Initialize the "doesn't return" information for each definition
           for (Defns ds=dsccs.head.getBindings(); ds!=null; ds=ds.next) {
               ds.head.resetDoesntReturn();
           }

           // Apply return analysis to each of the items in this scc.
           boolean changed = true;
           do {
               changed = false;
               for (Defns ds=dsccs.head.getBindings(); ds!=null; ds=ds.next) {
                    changed |= ds.head.returnAnalysis();
               }
           } while (changed);
      }
  }
}



/** Apply return analysis to this definition, returning true if this
 *  results in a change from the previously computed value.
 */
boolean returnAnalysis()
  case Defn { return false; }
  case Block {
      boolean newDoesntReturn = code.doesntReturn();
      if (newDoesntReturn != doesntReturn) {
          doesntReturn = newDoesntReturn;
          return true; // signal that a change was detected
      }
      return false; // no change
  }

/** Test for code that is guaranteed not to return.
```

```
 */
boolean doesntReturn()
case Code abstract;
case Done { return t.doesntReturn(); }
case Bind { return t.doesntReturn() || c.doesntReturn(); }
case Match {
    // If the default or any of the alternatives can return,
    // then the Match might also be able to return.
    if (def!=null && !def.doesntReturn()) {
        return false;
    }
    for (int i=0; i<alts.length; i++) {
        if (!alts[i].doesntReturn()) {
            return false;
        }
    }
    return true;
}
case TAlt      { return bc.doesntReturn(); }
case Tail      { return false; }
case BlockCall { return b.doesntReturn(); }
case PrimCall  { return p.doesntReturn(); }
case Prim      { return doesntReturn; }
```

2.4.2 Pre-Inlining Code Cleanup

```
void cleanup()
  case Defn  { /* Nothing to do here */ }
  case Block { code = code.cleanup(this); }

/** Perform simple clean-up of a code sequence before we begin the main
 *  inlining process.  If this ends up distilling the code sequence to
 *  a single Done, then we won't attempt inlining for this code (because
 *  anyone that attempts to use/call this block will be able to inline
 *  that call, and then this block will go away.
 */
Code cleanup(Block src)
  case Code { return this; }
  case Bind {
    if (v==Wildcard.obj && t.isPure()) {
      // Rewrite an expression (_ <- p; c) ==> c, if p is pure
      MILProgram.report("inlining eliminated a wildcard binding in " + src.getId());
      return c.cleanup(src);
    } else if (c.isReturn(v)) {
      // Rewrite an expression (v <- t; return v) ==> t
      MILProgram.report("applied right monad law in " + src.getId());
      return new Done(t);
    } else if (t.doesntReturn()) {
      // Rewrite (v <- t; c) ==> t, if t doesn't return
      MILProgram.report("removed code after a tail that does not return in " + src.getId());
      return new Done(t);
    } else {
      c = c.cleanup(src);
      return this;
    }
  }

/** Test whether a given Code/Tail value is an expression of the form return v,
 *  with the specified variable v as its parameter.  We also return a true result
 *  for a Tail of the form return _, where the wildcard indicates that any
 *  return value is acceptable because the result will be ignored by the caller.
 *  This allows us to turn more calls in to tail calls when they occur at the end
 *  of "void functions" that do not return a useful result.
 */
boolean isReturn(Var v)
```

```
case Code, Tail { return false; }
case Done       { return t.isReturn(v); }
case Return     { return a==v || a==Wildcard.obj; }
```

2.4.3 Detecting Loops

If a block `b` is defined by an equation of the form `b(x) = b(x)`, then any attempt to call `b` will result in an infinite loop. Even worse, if `b` is defined by an equation `b(x) = (v <- b(x); c)`, then any call to `b` will result in an infinite loop that is likely to be prematurely terminated by a stack overflow. More complex examples of this behavior occur in mutually recursive blocks, such as `b(x) = b1(x); b1(y) = (v <- b(y); b2(y))`. Although we do not expect examples like this to be particularly useful in practice, it is still possible for them to occur in MIL code, and they require special treatment during inlining to avoid sending the optimizer in to an infinite loop.

This section describes a simple algorithm for finding instances of these kinds of loops in MIL code, and, if a looping block is found, replacing its associated code sequence with a call to a `loop` primitive. For the examples above, this would mean rewriting the definitions of blocks `b` and `b1` to read as follows: `b(x) = loop()`, and `b1(y) = loop()`. Note that the `loop` primitive used here can be implemented directly by sending the program into an infinite loop. However, if appropriate, it can also be implemented by terminating the program, perhaps with an appropriate user diagnostic. In this way, we preserve the semantics of the program, but break the recursion that is expressed in the MIL code, and avoid the potential for infinite loops during inlining.

Of course, we do not detect arbitrary looping computations. For example, if the block `b` is defined by `b(x) = (v <- b1(x); b(v))`, then we will preserve the loop, which may still be productive (i.e., have a visible side-effect) as a result of whatever computations are done in `b1(x)`. Our inliner avoids the possibility of entering an infinite loop with examples like this by refusing to inline a call to a given block from a code sequence in the same strongly-connected component.

The loop detection algorithm works by visiting each block `b` in the program, and testing to see if the associated code sequence begins with a call to another block (either directly through `Done`, or through the `Tail` part of a `Bind`). If there is no initial `BlockCall`, or if there is a call to a block in a different (and hence earlier) strongly-connected component, then there is no looping behavior. Otherwise, if the code starts with a call to a new block, `b'`, then we add `b` to a list of distinct `Blocks` that have been `visited` so far, and start testing to determine whether `b'` will result in a loop. If this process continues long enough to arrive back at a `Block` that is already included in the `visited` list, then we have found a loop, and can rewrite all of the blocks in `visited` with a direct call to the `loop` primitive. Otherwise, we can conclude that the original block `b` does not fit the pattern of an immediate loop that would send the subsequent inlining process in to an infinite loop.

```
macro AddIsIn(Block) // Add an implementation of isIn for the Blocks class

boolean detectLoops(Blocks visited)
  case Defn  { return false; }
  case Block {
    // Check to see if this block calls code for an already visited block:
    if (Blocks.isIn(this, visited) || code.detectLoops(this, visited)) {
      MILProgram.report("detected an infinite loop in block " + getId());
      code = new Done(PrimCall.loop);
      return true;
    }
    return false;
  }

boolean detectLoops(Block src, Blocks visited)
  case Code, Tail { return false; }
  case Done {     // look for src(x) = b(x)
    return t.detectLoops(src, visited);
  }
  case Bind {     // look for src(x) = (v <- b(x); ...), possibly with some
                  // initial prefix of pure bindings x1 <- pt1; ...; xn <- ptn
    return t.detectLoops(src, visited)
        || (t.isPure() && c.detectLoops(src,visited));
  }
  case BlockCall  {     // Keep searching while we're still in the same SCC
    return (src.getScc()==b.getScc())
        && b.detectLoops(new Blocks(src, visited));
  }
```

We end this section with the declarations for the `loop` primitive that is used in `detectLoops()`.

```
class Prim {
  /** Represents a primitive that will enter a non-terminating loop.  Marked as
   *  impure because eliminating a call to this primitive, even when its value
   *  is not used will typically change the behavior of the enclosing program.
   */
  public static final Prim loop = new Prim("loop", 0, Prim.IMPURE, true);
}
class PrimCall {
  public static final Call loop = new PrimCall(Prim.loop).withArgs(new Atom[0]);
}
```

Of course, the implementation of `loop`, like any other primitive, must still be provided somewhere in the runtime system. With the intended semantics, however, it would also be possible to apply further rewrites to code sequences involving calls to `loop` so that, for example, `(v <- loop(()); c)` can be replaced directly by the primitive call `loop(())`. We return to tackle this in Section **??**.

```
// TODO: Not used!
boolean loops()
  case Tail     { return false; }
  case PrimCall { return p==Prim.loop; }
```

2.4.4 Inlining Within Code Sequences

```
/** Apply inlining to the code in this definition.
 */
public void inlining()
```

```
  case Defn abstract;
  case Block       {
    code = code.inliningBody(this);
  }
  case TopLevel, ClosureDefn    {
       tail = tail.inlineTail();
  }

class Code {
  public static final int INLINE_ITER_LIMIT = 3;
}

/** Perform inlining on this Code sequence, looking for opportunities
 *  to: inline BlockCalls in both Bind and Done nodes; to skip goto
 *  blocks referenced from Match nodes; and to apply the right monad
 *  law by rewriting Code of the form (v <- t; return v) as t.
 *  As a special case, we do not apply inlining to the code of a block
 *  that contains a single Tail; any calls to that block should be
 *  inlined anyway, at which point the block will become deadcode.
 *  In addition, expanding a single tail block like this could lead
 *  to significant code duplication, not least because we also have
 *  another transformation (toBlockCall) that turns an expanded code
 *  sequence back in to a single block call, and we want to avoid
 *  oscillation back and forth between these two forms.
 *
 *  TODO: Alas, the special treatment of TopLevel values doesn't work.
 *  A code sequence of the form (_ <- return 0; b()) for a top-level
 *  value can trick the inliner into firing (because it's not a single
 *  Done value), expanding the code for b() in place.  Forcing every
 *  TopLevel to be a BlockCall doesn't work either because then the
 *  code that inlines enters to top-level closures won't notice that
 *  the top-level value is indeed a closure (because it goes through
 *  an unnecessary block).  This could probably be handled by a special
 *  case that routes through the block to see if a simple closure can be
 *  found ...
 */
Code inliningBody(Block src)
  case Code { return inlining(src, INLINE_ITER_LIMIT); }

/** Perform inlining on this Code value, decrementing the limit each time
 *  a successful inlining is performed, and declining to pursue further
 *  inlining at this node once the limit reachs zero.
 */
Code inlining(Block src, int limit)
  case Code abstract;
  case Done {
    if (limit>0) {  // Is this an opportunity for suffix inlining?
      Code ic = t.suffixInline(src);
      if (ic!=null) {
        return ic.inlining(src, limit-1);
      }
    }
    return this;
  }
  case Bind {
    if (limit>0) {  // Is this an opportunity for prefix inlining?
      Code ic = t.prefixInline(src, v, c);
      if (ic!=null) {
        return ic.inlining(src, limit-1);
      }
    }
    c = c.inlining(src, INLINE_ITER_LIMIT);

    // Rewrite an expression (v <- b(x,..); invoke v) ==> b'(x,..)
    //                    or (v <- b(x,..); v @ a)    ==> b'(x,..,a)
    BlockCall bc = t.isBlockCall();
    if (bc!=null) {
      Code nc;
```

```
      if ((nc = c.invokes(v, bc))!=null) {
        MILProgram.report("pushed invoke into call in " + src.getId());
        return nc;
      }
      if ((nc = c.enters(v, bc))!=null) {
        MILProgram.report("pushed enter into call in " + src.getId());
        return nc;
      }
      if ((nc = c.casesOn(v, bc))!=null) {
        MILProgram.report("pushed case into call in " + src.getId());
        return nc;
      }

    }
    return this;
  }
  case Match {
    for (int i=0; i<alts.length; i++) {
      alts[i].inlineAlt();
    }
    if (def!=null) {
      def = def.inlineBlockCall();
    }
    // If the Match has no alternatives, then we can use the default directly.
    return (alts.length==0) ? new Done(def).inlining(src, INLINE_ITER_LIMIT) : this;
    // TODO: is it appropriate to apply inlining to the def above?  If there is useful
    // work to be done, we'll catch it next time anyway ... won't we ... ?
  }

void inlineAlt()
  case TAlt { bc = bc.inlineBlockCall(); }
```

2.4.5 Prefix Inlining

We use the term 'prefix inlining' to refer to the transformation that replaces a block call in a code sequence of the form `(u <- b(x); d)` with a copy of the body of `b(x)`. For example, if `b(x) = (u1 <- t1; ...; un <- tn; t)`, then we can rewrite:

```
(u <- b(x); d) = (u <- (u1 <- t1'; ...; un <- tn'; t'); d)
               = (u1 <- t1'; ...; un <- tn'; u <- t'; d)
```

The first equality shown here is a direct substitution, replacing the call `b(x)` with its body. Technically speaking, the resulting expression, with nested binds, is not a valid MIL program. However, we can rewrite the program to fit the syntax of MIL by using the monad associativity law, as shown in the second equality.

In practice, a slightly more complex tranformation is needed to substitute the actual arguments appearing in the call to the block for the formal parameters that are used in its definition. At the same time, it is also prudent to replace the variables `u1`, ..., `un` from the definition of the block with fresh variables; this avoids the potential for inadvertent capture problems. For these reasons, the following implementation of the prefix inlining transformation also includes a substitution argument, initialized to describe the mapping of actuals to formals, in the original call, and suitably extended as we visit each `Bind`.

```
Code prefixInline(AtomSubst s, Var u, Code d)
```

```
  case Code {
    debug.Internal.error("This code cannot be inlined");
    return this;
  }
  case Done {
    return new Bind(u, t.apply(s), d);
  }
  case Bind {
    Var w = new Temp();
    return new Bind(w, t.apply(s), c.prefixInline(new AtomSubst(v, w, s), u, d));
  }
```

Uses of prefix inlining must be carefully controlled. For example, it is only defined for cases where the code for the block that is being inlined is a sequence of `Bind`s ending in a `Done`. As indicated in the catch-all case above for `Code`, any attempt to apply this transformation with a code sequence that does not have the right form will lead to an error. Use of prefix inlining must also be limited so that we only inline the definition of a block when it is called from a different strongly-conected component; this is necessary to avoid nontermination problems. Finally, to avoid increasing the size of the program after inlining with unnecessarily duplicated code, it is prudent to limit the use of prefix inlining to blocks that have a relatively short code sequence. Specifically, we can test to determine if prefix inlining can be used for a call to a block `b` from code in a source block `src` by using the call `b.canPrefixInline(src)`:

```
class Block {
  public static final int INLINE_LINES_LIMIT = 4;

  boolean canPrefixInline(Block src) {
    if (this.getScc()!=src.getScc()) {         // Restrict to different SCCs
      int n = code.prefixInlineLength(0);
      return n>0 && (occurs==1 || n<=INLINE_LINES_LIMIT);
    }
    return false;
  }
}
```

This definition uses an auxiliary method, `prefixInlineLength()` to compute the length of a code sequence. The argument tracks the number of `Bind` nodes that have already been passed (it should be `0` when the method is first called), while the result is either the length of the code sequence (counting one for each `Bind` and each `Done` node) or `0` if the code sequence ends with something other than `Done`.

```
int prefixInlineLength(int len)
  case Code { return 0; }
  case Done { return len+1; }
  case Bind { return c.prefixInlineLength(len+1); }


Code prefixInline(Block src, Var r, Code c)
  case Tail      { return null; }
  case BlockCall { return b.prefixInline(src, args, r, c); }

class Block {
  /** Attempt to inline the code for this block onto the front of another
```

```
   *  block of code.  Assumes that the final result computed by this block
   *  will be bound to the variable r, and that the computation will proceed
   *  with the code specified by rest.  The src value specifies the block
   *  in which the original BlockCall appeared while args specifies the set
   *  of arguments that were passed in at that call.  A null return indicates
   *  that no inlining was performed.
   */
  Code prefixInline(Block src, Atom[] args, Var r, Code rest) {
    if (canPrefixInline(src)) {
      MILProgram.report("prefixInline succeeded for call to block " + getId()
                        + " from block " + src.getId());
      return code.prefixInline(AtomSubst.extend(formals, args, null), r, rest);
    }
    return null;
  }
}
```

2.4.6 Suffix Inlining

We use the term 'suffix inlining' to refer to the transformation that replaces a block call in the tail position of a code sequence (i.e., in a `Done` node) with a copy of the body of the block. For example, if `b(x) = (u <- t; c)`, then we can rewrite:

```
(x <- b1(y); b(x))  ==  (x <- b1(y); u <- t; c)
```

This is a straightforward rewrite, although in general we will need to apply a substitution to the body of `b(x)` that replaces each of the formal parameters of `b` with the corresponding actual parameter from the call, as in the following variant of the previous example that makes use of a renaming substitution `[z/x]` to replace free occurrences of `x` with `z`:

```
(z <- b1(y); b(z))  ==  (z <- b1(y); u <- [z/x]t; [z/x]c)
```

To avoid unintended name capture issues, it is also safest to replace each of the variables introduced in the body of the block that we are inlining (such as `u` in the example shown here) with fresh variables. Fortunately, our implementation of substitution on code sequences (Section 1.5.4) already takes care of this, which makes it very easy for us to give the following implementation for suffix inlining:

```
/** Perform suffix inlining on this tail, which either replaces a block call
 *  with an appropriately renamed copy of the block's body, or else returns
 *  null if the tail is either not a block call, or if the code of the block
 *  is not suitable for inlining.
 */
Code suffixInline(Block src)
  case Tail      { return null; }
  case BlockCall { return b.suffixInline(src, args); }

/** Attempt to construct an inlined version of the code in this block that can be
 *  placed at the end of a Code sequence.  Assumes that a BlockCall to this block
 *  with the given set of arguments was included in the specified src Block.  A
 *  null return indicates that no inlining was performed.
```

```
 */
Code suffixInline(Block src, Atom[] args)
  case Block {
      if (canSuffixInline(src)) {
          MILProgram.report("suffixInline succeeded for call to block " + getId()
                            + " from block " + src.getId());
          return code.apply(AtomSubst.extend(formals, args, null));
      }
      return null;
  }
```

As with prefix inlining, the real challenge is in determining when it is appropriate to inline: Judicious use of inlining might help to reduce the overheads of function calls, and to expose new opportunities for other optimizations. On the other hand, overly aggressive inlining might lead to significant and unproductive code duplication.

```
/** We allow a block to be inlined if the original call is in a different
 *  block, the code for the block ends with a Done, and either there is
 *  only one reference to the block in the whole program, or else the
 *  length of the code sequence is at most INLINE_LINES_LIMIT lines long.
 */
boolean canSuffixInline(Block src)
  case Block {
      if (occurs==1 || code.isDone()!=null) {    // Inline single occurrences and trivial
          return true;                           // blocks (safe, as a result of removing loops)
      } else if (this.getScc()==src.getScc()) {  // But otherwise don't inline blocks in same SCC
          return false;
      } else {
          int n = code.suffixInlineLength(0);    // Inline short code blocks
          return n>0 && n<=INLINE_LINES_LIMIT;
      }
  }

/** Compute the length of this Code sequence for the purposes of prefix inlining.
 *  The returned value is either the length of the code sequence (counting
 *  one for each Bind and Done node) or 0 if the code sequence ends with
 *  something other than Done.  Argument should be initialized to 0 for
 *  first call.
 */
int suffixInlineLength(int len)
  case Code  abstract;
  case Match { return len+1; }  // TODO: maybe pick a higher value?
  case Done  { return len+1; }
  case Bind  { return c.suffixInlineLength(len+1); }
```

2.4.7 Inlining Tails

```
/** Skip goto blocks in a Tail (for a ClosureDefn or TopLevel).
 *  TODO: can this be simplified now that ClosureDefns hold Tails rather than Calls?
 *
 *  TODO: couldn't we return an arbitrary Tail here, not just a Call?
 */
public Tail inlineTail()
  case Tail { return this; }
  case Enter {
    // TODO: uses code from later flow analysis section
    // TODO: duplicates code from rewrite for Enter
    // TODO: do we need a parallel case for Invoke?
    ClosAlloc clos = f.lookForClosAlloc(null); // Is f a known closure?
    if (clos!=null) {
```

```
      MILProgram.report("rewriting "+f+" @ "+a);
      return clos.withArg(a);                // If so, apply it in place
    }
    return this;
  }
  case BlockCall {
    BlockCall bc   = this.inlineBlockCall();
    Tail      tail = bc.b.inlineTail(bc.args);
    return (tail==null) ? bc : tail;
  }

/** Test to see if a call to this block with specific arguments can be
 *  replaced with a Call.
 */
public Tail inlineTail(Atom[] args)
  case Block {
    Tail tail = code.isDone();
    if (tail!=null) {
      tail = tail.forceApply(AtomSubst.extend(formals, args, null));
    }
    return tail;
  }

/** Test to determine whether this Code is a Done.
 */
public Tail isDone()
  case Code { return null; }
  case Done { return t; }
```

2.4.8 Rewrite BlockCalls to Skip Goto Blocks

If the definition of a block `b` looks like `b(x..) = b1(a..)` for some other block `b1` (i.e., `b /= b1`), then a `BlockCall b(c..)` that is mentioned in a `TAlt` or in the default for a `Match` can be replaced with `[c../x..]b1(a..)`. We'll refer to a block like `b(x..) = b1(a..)` as a "goto block".

```
/** Rewrite a BlockCall, if possible, to skip over goto blocks.
 */
public BlockCall inlineBlockCall()
  case BlockCall {
    BlockCall bc = b.inlineBlockCall(args);
    if (bc!=null) {
      MILProgram.report("eliminateGoto succeeded on call to "+bc.b.getId());
      return bc;
    }
    return this;
  }

/** Test to determine whether this is a goto block.  The specified list of
 *  arguments indicates the parameter list that is used on entry to this
 *  block.
 */
public BlockCall inlineBlockCall(Atom[] args)
  case Block {
    BlockCall bc = code.isGotoBlockCode(this);
    if (bc!=null) {
      bc = bc.forceApplyBlockCall(AtomSubst.extend(formals, args, null));
    }
    return bc;
  }

/** Test to determine whether this Code/Tail in a specified src Block
 *  constitutes a goto block.  For this to be valid, the code must be
```

```
 *  an immediate BlockCall to a different block.
 */
public BlockCall isGotoBlockCode(Block src)
  case Code, Tail { return null; }
  case Done       { return t.isGotoBlockCode(src); }
  case BlockCall  { return (b!=src) ? this : null; }
```

2.5 Lifting Allocators

In this section, we describe a transformation that modifies `Code` sequences to move allocators towards the front of the sequence by using the following rewrite:

```
(x<-nonalloc; v<-alloc; c)  ==  (v<-alloc; x<-nonalloc; c)
```

The assumptions here are that: `alloc` is an allocator (i.e., an instance of `DataAlloc`, `ClosAlloc`, or `CompAlloc`—the subclasses of `Allocator`); `nonalloc` is a non-allocator (i.e., any other form of `Tail` expression); `v` and `x` are distinct variables; `x` does not appear in `alloc`; and `v` does not appear in `nonalloc`. It is easy, of course, to implement a test that distinguishes between allocators and non-allocators.

```
public Allocator isAllocator()
  case Tail      { return null; }
  case Allocator { return this; }
```

The distinction is useful because allocators have only a very limited form of side-effect: an allocator modifies the heap but does not produce a user visible side-effect. As such, switching the order of an allocator and a non-allocator, as in the rewrite, is valid because it will not produce a visible change in program behavior[1]. This transformation has the potential to group allocators together, which can be useful during code generation where it might allow the cost of a heap exhaustion check to be shared across multiple allocators. In addition, this transformation can also bring previously separated non-allocators together and so enable the use of other optimizations.

Each of the three columns in the following example shows a different variant of a particular code sequence.

```
f <- b(y,z)          v <- k{u}           v <- k{u}
v <- k{u}            f <- b(y,z)         p <- b'(y,z,a)
p <- f @ a           p <- f @ a          c
c                    c
```

[1] To fully justify this assumption, we need either to guarantee that there will never be any out of memory errors (so allocation never fails), or else use a semantics that identifies an out of memory condition with nontermination.

The version on the left can be transformed to the version in the middle by moving the use of an allocator (`k{u}`) forward in the code sequence and bringing the call to block `b` next to the instruction where its result is entered with argument `a`. If `f` is not used in the rest of the code, `c`, then we can replace the call to `b` with a call to a transformed variant `b'` that takes the argument `a` as an extra parameter, obtaining the version in the right column. This can enble extra optimizations to be performed in the code for `b'`, derived from the code for `b`, because it makes the argument accessible within `b'`.

Moving the allocator forward in this way makes it easier to spot the pattern of a function value being returned from a block and immediately entered. However, it does have the potential to increase register pressure: the value of `v` must be maintained across the call to `b'(y,z,a)`, for example. Another possibility in the example above that does not increase register pressure would be to apply our rewrite in reverse, moving the allocator after the enter instruction that appears in Line 3 of the original. But if we change the example a little so that the result of the allocator is used in Line 3, then it is no longer possible to delay the allocator, but we can still bring it forward in the same way as the preceding example:

```
f <- b(y,z)          a <- k{u}           a <- k{u}
a <- k{u}            f <- b(y,z)         p <- b'(y,z,a)
p <- f @ a           p <- f @ a          c
c                    c
```

For this reason, we prefer to focus here on the more general technique of moving allocators forward, and to consider subsequent rewriting of blocks to improve register allocation later as a separate concern.

The implementation of this transformation begins again with a loop that visits all of the definitions in the current program (although only `Block`s are relevant here because they are the only kind of definition that contain a code sequence).

```
class MILProgram {
  /** Run a liftAllocators pass over this program, assuming a previous
   *  call to shake() to compute SCCs.
   */
  public void liftAllocators() {
    for (DefnSCCs dsccs = sccs; dsccs!=null; dsccs=dsccs.next) {
      for (Defns ds=dsccs.head.getBindings(); ds!=null; ds=ds.next) {
        ds.head.liftAllocators();
      }
    }
  }
}

void liftAllocators()
  case Defn  { /* Nothing to do */ }
  case Block { code.liftAllocators(); }
```

The rewriting process itself is implemented by a pair of `liftAllocator()` methods. The first, which does not take a parameter, covers the general case where there are no assumptions about the context in which the `Code` sequence appears.

```
void liftAllocators()
  case Code { /* Nothing to do in this case */ }
  case Bind {
    if (t.isAllocator()!=null) {
      // This bind uses an allocator, so we can only look for lifting opportunities in
      // the rest of the code.
      c.liftAllocators();
    } else {
      // This bind does not have an allocator, but it could be used as a non-allocator
      // parent for the following code ... which might turn this node into an allocator,
      // prompting the need to repeat the call to this.liftAllocators().
      //
      if (c.liftAllocators(this)) {
        this.liftAllocators();
      }
    }
  }
```

The second `liftAllocators()` method is used only for the special case where the given code sequence appears as an immediate tail of a `Bind` of the form `v <- nonalloc`, where `nonalloc` is a non-allocator. In this case, a single parameter is used, pointing directly to the `Bind` immediately above the current `Code` value. When this method is used on a code sequence that begins with a `Bind` for an allocator, then we may have an opportunity to apply the rewrite and switch the order of the two `Bind`s. (Or rather, to swap the components of the two `Bind`s.) The final result is a Boolean; a `true` value indicates that a modification was made.

```
boolean liftAllocators(Bind parent)
  case Code { return false; }
  case Bind {
    if (t.isAllocator()!=null) {
      // This bind uses an allocator, so it can be swapped with the parent Bind,
      // if that is safe.
      if (this.v!=parent.v
          && !this.t.contains(parent.v)
          && !parent.t.contains(this.v)) {
        Var  tempv = parent.v; parent.v = this.v; this.v = tempv;  // swap vars
        Tail tempt = parent.t; parent.t = this.t; this.t = tempt;  // swap tails
        MILProgram.report("lifted allocator for " + parent.v);
        c.liftAllocators(this); // Now this node is a non-allocator parent of c.
        return true;
      }

      // We can't change this Bind, but can still scan the rest of the code:
      c.liftAllocators();
      return false;
    } else {
      // This bind does not have an allocator, but it could be used as a non-allocator
      // parent for the following code ... if that changes this node, then we might
      // have a second opportunity to rewrite this node, hence the tail call:
      return c.liftAllocators(this) && this.liftAllocators(parent);
    }
  }
```

2.6 Eliminating Unused Arguments

This section describes an analysis and associated program transformation that detects and eliminates unused formal parameters in block definitions and unused stored fields in closure definitions. In the following program fragment, for example, we can see that the `y` parameter is not used in the code for block `b`:

```
b(x,y) = t <- add((x,x)); add((t,x))
k{y} x = b(x, y)
```

Given this observation, we can rewrite the definition of `b` to omit the formal parameter `y`—so long as we also rewrite every call to `b` to omit the corresponding argument. For the current example, this results in the following updated code:

```
b(x)   = t <- add((x,x)); add((t,x))
k{y} x = b(x)
```

Now, of course, it is obvious that there is no need to store the `y` field in the closure for `k` because it will not be needed when the body, `b(x)`, of the closure is executed. As such, we can further simplify the code for our example to the following form:

```
b(x)  = t <- add((x,x)); add((t,x))
k{} x = b(x)
```

The techniques that we describe in this section also work with slightly more tricky examples that can occur when dealing with recursive definitions. In the following code, for example, the `z` argument for the block `b0` is never actually used, despite the fact that it appears as a free variable on the right hand side of the block definition:

```
b0(z) = t <- b1(); _ <- b2(t); b0(z)
```

Although they are not very common, definitions like these do occur in practice. Eliminating unused arguments in such cases is worthwhile because it can simplify the generated code, avoid unnecessary overhead for parameter passing, ease the task of register allocation, and reduce the storage requirements for closure data structures. Examples like the ones shown above confirm that, in general, we will need to use an iterative data flow analysis to detect unused arguments: we assume initially that no arguments are use; we analyze each code sequence in the program and adjust our assumptions when we encounter arguments that are used (such as the `x` parameter in the preceding definition for `b`); and we repeat this process until it reaches a fixed point (for example, we never find any uses of `z` in the definition of `b0` to suggest that it might be needed).

In the rest of this section, we will describe a two phase algorithm. The first phase uses a traditional data flow analysis to determine which arguments of each block and closure definition are used. The second phase consists of a program transformation pass that uses the results of the analysis to rewrite the program, eliminating unnecessary arguments on both the left and right hand sides of each definition as appropriate. The results of the analysis, and the inputs to the rewriting phase, are captured by the following fields that are included in each `Block` and `ClosureDefn` object. In essence, these fields allow us to represent the subset of the argument values for each block or closure definition that are actually used. As suggested previously, we initialize these fields to a representation of the empty set, and then let them grow as we determine which arguments are actually needed.

```
class Block, ClosureDefn {  // TODO: consider introducing common subclass?
  /** A bitmap that identifies the used arguments of this definition.  The base case,
   *  with no used arguments, can be represented by a null array.  Otherwise, it will
   *  be a non null array, the same length as the list of true in positions corresponding
   *  to arguments that are known to be used and false in all other positions.
   */
  private boolean[] usedArgs = null;

  /** Counts the total number of used arguments in this definition; this should match
   *  the number of true bits in the usedArgs array.
   */
  private int numUsedArgs = 0;
}
```

The overall algorithm for detecting and removing unused arguments is captured by the following `eliminateUnusedArgs()` method:

```
/** Analyze and rewrite this program to remove unused Block and ClosureDefn arguments.
 */
void eliminateUnusedArgs()
  case MILProgram {
      int totalUnused = 0;

      // Calculate unused argument information for every Block and ClosureDefn:
      for (DefnSCCs dsccs = sccs; dsccs!=null; dsccs=dsccs.next) {
           // Initialize used argument information for each definition
           for (Defns ds=dsccs.head.getBindings(); ds!=null; ds=ds.next) {
               ds.head.clearUsedArgsInfo();
           }

           // Calculate the number of unused arguments for the definitions in this
           // strongly-connected component, iterating until either there are no
           // unused arguments, or else until we reach a fixed point (i.e., we get
           // the same number of unused args on two successive passes).
           int lastUnused = 0;
           int unused     = 0;
           do {
               lastUnused = unused;
               unused     = 0;
               for (Defns ds=dsccs.head.getBindings(); ds!=null; ds=ds.next) {
                    unused += ds.head.countUnusedArgs();
               }
           } while (unused>0 && unused!=lastUnused);
           totalUnused += unused;
      }
```

```
        // Rewrite the program if there are unused arguments:
        if (totalUnused>0) {
            for (DefnSCCs dsccs = sccs; dsccs!=null; dsccs=dsccs.next) {
                 for (Defns ds=dsccs.head.getBindings(); ds!=null; ds=ds.next) {
                      ds.head.removeUnusedArgs();
                 }
            }
        }
    }
```

The analysis phase of the algorithm (corresponding to the main `for` loop in `eliminateUnusedArgs()`), works in sequence through each of the strongly-connected components of the program. This is useful in practice because it breaks down the task of analyzing a full program in to the task of analyzing a sequence of (typically much smaller) strongly-connected components, which can reduce the overall cost of the analysis. This part of the algorithm begins by using `clearUsedArgsInfo()` to set the `unusedArgs` and `numUnusedArgs` fields to appropriate initial values, and then uses the `countUnusedArgs()` method on each of the definitions in the current components to look for unused arguments. The latter process, captured in the code by a `do-while` loop, terminates either if it determines that there are no unused arguments, or else when there is no change in the unused argument count from one iteration to the next. Further details about the analysis phase are provided in Section 2.6.1.

The rewriting phase of the algorithm uses a family of `removeUnusedArgs()` methods to traverse the abstract syntax tree of a complete program, updating any definitions and calls that have unused arguments. In this case, the iteration through strongly-connected components is just an consequence of the way that MIL programs are represented, and it is not essential to the algorithm; we would obtain the same final result by visiting the individual parts of the abstract syntax tree in any other order. The rewriting phase is only executed in the case where `totalUnused>0`—indicating that the analysis was able to discover at least one instance of an unused argument—because otherwise rewriting the program would not have any effect. Further details about the rewriting phase are provided in Section 2.6.2.

2.6.1 Detecting Unused Arguments

As described previously, the analysis phase—which is responsible for detecting unused arguments—begins by assuming that none of the arguments of any block or closure definition are used. In practice, this means that the first step in the analysis is to set `usedArgs` to `null` and `numUsedArgs` to `0`, which corresponds to a representation for an empty set (of variables).

```
/** Reset the bitmap and count for the used arguments of this definition,
 *  where relevant.
 */
void clearUsedArgsInfo()
  case Defn abstract;
  case TopLevel { /* nothing to do in this case */ }
```

```
  case Block, ClosureDefn {
      usedArgs    = null;
      numUsedArgs = 0;
  }
```

The key step in the analysis process is the attempt to determine the number of unused arguments in a given block or closure definition. (We refer to this as an 'attempt' because the answers that we get may change as we analyze more of the program.) The calculation is based on a comparison of the list of formal parameters (for a `Block`, or the list of stored fields for a `ClosureDefn`) with the list of variables that are actually used on the right hand side of the definition. In all cases, we assume that the latter is a (not-necessarily strict) subset of the former because we do not allow unbound variables in MIL programs. In particular, because `TopLevel` definitions do not have any parameters, they can only reference other values that are defined at the top level, and cannot have any free variables.

```
/** Count the number of unused arguments for this definition using the current
 *  unusedArgs information for any other items that it references.
 */
int countUnusedArgs()
  case Defn abstract;
  case TopLevel    { return 0; }
  case Block       { return countUnusedArgs(formals); }
  case ClosureDefn { return countUnusedArgs(stored); }
```

Of course, if we have previously determined that all of the arguments of a given definition are used (which includes, as a special case, definitions that have an empty list of arguments), then there is nothing more to be done. But otherwise, we can use an appropriate `usedVars()` method (described in more detail below) to determine the set of variables, `vs`, that are actually used on the right hand side of the definition, and then ensure that every argument appearing in `vs` has the corresponding flag set in `usedArgs`.

```
/** Count the number of unused arguments for this definition.  A count of zero
 *  indicates that all arguments are used.
 */
int countUnusedArgs(Var[] bound)
  case Block, ClosureDefn {
      int unused = bound.length - numUsedArgs;      // count # of unused args
      if (unused > 0) {                             // skip if no unused args
          Vars vs = usedVars();                     // find vars used in body
          for (int i=0; i<bound.length; i++) {      // scan argument list
              if (usedArgs==null || !usedArgs[i]) { // skip if already known to be used
                  if (Vars.isIn(bound[i], vs) && !duplicated(i, bound)) {
                      if (usedArgs==null) {         // initialize usedArgs for first use
                          usedArgs = new boolean[bound.length];
                      }
                      usedArgs[i] = true;           // mark this argument as used
                      numUsedArgs++;                // update counts
                      unused--;
                  }
              }
          }
      }
      return unused;
  }
```

Note that we only mark a parameter as used if, first, that parameter is included in the set of used variables (i.e., if `Vars.isIn(bound[i], vs)`) and, second, if there are no other uses of the same variable at a lower index in the argument list (i.e., if `!duplicated(i, bound)`). We do not expect this latter condition to be a common occurrence, but still check for it, using the following definition for the `duplicated()` method, to allow for the possibility that such examples might be introduced by other transformations within the optimizer or code generator.

```
/** A utility function that returns true if the variable at position i
 *  in the given array also appears in some earlier position in the array.
 *  (If this condition applies, then we can mark the later occurrence as
 *  unused; there is no need to pass the same variable twice.)
 */
static private boolean duplicated(int i, Var[] bound)
  case Block, ClosureDefn {
      // Did this variable appear in an earlier position?
      for (int j=0; j<i; j++) {
          if (bound[j]==bound[i]) {
              return true;
          }
      }
      return false;
  }
```

We define a family of `usedVars()` methods for computing the set of variables that are used in a given `Block`, `ClosureDefn`, `Code`, or `Tail` object.

```
/** Find the list of variables that are used in this definition.  Variables that
 *  are mentioned in BlockCalls, ClosAllocs, or CompAllocs are only included if
 *  the corresponding flag in usedArgs is set.
 */
Vars usedVars()
  case Block       { return code.usedVars(); }
  case ClosureDefn { return tail.usedVars(null); }

/** Find the list of variables that are used in this code sequence.  Variables
 *  that are mentioned in BlockCalls, ClosAllocs, or CompAllocs are only
 *  included if the corresponding flag in usedArgs is set.
 */
Vars usedVars()
  case Code  abstract;
  case Done  { return t.usedVars(null); }
  case Bind  { return t.usedVars(Vars.remove(v, c.usedVars())); }
  case Match {
    Vars vs = (def==null) ? null : def.usedVars(null);
    for (int i=0; i<alts.length; i++) {
      vs = Vars.add(alts[i].usedVars(), vs);
    }
    return a.add(vs);
  }
  case TAlt  { return Vars.remove(args, bc.usedVars(null)); }

/** Find the variables that are used in this Tail expression, adding them to
 *  the list that is passed in as a parameter.  Variables that are mentioned
 *  in BlockCalls, ClosAllocs, or CompAllocs are only included if the
 *  corresponding flag in usedArgs is set; all of the arguments in other types
 *  of Call (i.e., PrimCalls and DataAllocs) are considered to be "used".
 */
Vars usedVars(Vars vs)
  case Tail abstract;
  case Return, Invoke { return a.add(vs); }
  case Enter          { return f.add(a.add(vs)); }
```

```
  case BlockCall      { return b.usedVars(args, vs); }
  case ClosAlloc      { return k.usedVars(args, vs); }
  case CompAlloc      { return m.usedVars(args, vs); }
  case Call {
    for (int i=0; i<args.length; i++) {
      vs = args[i].add(vs);
    }
    return vs;
  }
```

The key point in these definitions is that arguments appearing in a `BlockCall`, a `ClosAlloc`, or a `CompAlloc` are only considered to be used if the `usedArgs` flag for the corresponding parameter has been set. In other words, arguments appearing in positions for which the corresponding `usedArgs` entry is `false` (or for which `usedArgs` is `null`) are not added to the `Vars` list produced by the `usedVars()` methods. The necessary tests for this condition are captured in the following (and final) variation of `usedVars()`:

```
/** Find the list of variables that are used in a call to this definition,
 *  taking account of the usedArgs setting so that we only include variables
 *  appearing in argument positions that are known to be used.
 */
Vars usedVars(Atom[] args, Vars vs)
  case Block, ClosureDefn {
    if (usedArgs!=null) {       // ignore this call if no args are used
      for (int i=0; i<args.length; i++) {
        if (usedArgs[i]) {      // ignore this argument if the flag is not set
          vs = args[i].add(vs);
        }
      }
    }
    return vs;
  }
```

2.6.2 Removing Unused Arguments

Once the set of used arguments for each block and closure definition has been computed, we can update the program to eliminate any unused arguments. The main challenge here is simply to make sure that we apply the same rewrites consistently throughout the whole program. For example, if we remove a particular parameter from the definition of a given block, then we must also remove the corresponding argument value in every call to that block.

The following pair of methods capture the key steps in using the information in a `usedArgs` array to trim down an array of variables (representing the `formals` or `stored` values in a `Block` or `ClosureDefn`, respectively) and an array of atoms (representing the `args` in a `BlockCall`, `ClosAlloc`, or `CompAlloc`):

```
class Block, ClosureDefn {

  /** Use information about which and how many argument positions are used to
   *  trim down an array of variables (i.e., the formal parameters of a Block
   *  or the stored fields of a ClosureDefn).
   */
  Var[] removeUnusedVars(Var[] vars) {
```

```
      if (numUsedArgs==vars.length) {   // All arguments used, no rewrite needed
          return vars;
      } else if (usedArgs==null) {      // Drop all arguments; none are needed
          MILProgram.report("removing all arguments from " + getId());
          return Var.noVars;
      } else {                          // Select the subset of needed arguments
          Var[] newVars = new Var[numUsedArgs];
          for (int i=0, j=0; i<vars.length; i++) {
              if (usedArgs[i]) {
                  newVars[j++] = vars[i];
              } else {
                  MILProgram.report("removing unused argument " + vars[i]
                                                          + " from " + getId());
              }
          }
          return newVars;
      }
  }

  /** Update an argument list by removing unused arguments.
   */
  void removeUnusedArgs(Call call, Atom[] args) {
      if (numUsedArgs!=args.length) {   // Only rewrite if some arguments are unused
         Atom[] newArgs = new Atom[numUsedArgs];
         for (int i=0, j=0; j<numUsedArgs; i++) {
             if (usedArgs[i]) {         // copy used arguments
                 newArgs[j++] = args[i];
             }
         }
         call.withArgs(newArgs);        // set new argument list for this call
      }
  }
}
```

All that remains is to visit each node in the abstract syntax tree and apply the `removeUnusedVars()` method to each left hand side and `removeUnusedArgs()` to each right hand side as appropriate. The following set of method definitions provides the appropriate treatment for each different form of `Defn`, `Code`, and `Tail` value.

```
/** Rewrite this program to remove unused arguments in block calls.
 */
void removeUnusedArgs()
  // Cases for definitions:
  case Defn abstract;
  case Block {
      formals = removeUnusedVars(formals); // remove unused formal parameters
      code.removeUnusedArgs();             // update calls in code sequence
  }
  case ClosureDefn {
      stored = removeUnusedVars(stored);   // remove unused stored parameters
      tail.removeUnusedArgs();             // update calls in tail
  }
  case TopLevel {
      tail.removeUnusedArgs();
  }

  // Cases for code sequences:
  case Code abstract;
  case Done { t.removeUnusedArgs(); }
  case Bind {
      t.removeUnusedArgs();
      c.removeUnusedArgs();
  }
```

```
case Match {
    if (def!=null) {
        def.removeUnusedArgs();
    }
    for (int i=0; i<alts.length; i++) {
        alts[i].removeUnusedArgs();
    }
}
case TAlt      { bc.removeUnusedArgs(); }

// Cases for Tail expressions:
case Tail      { /* nothing to do here */ }
case BlockCall { b.removeUnusedArgs(this, args); }
case ClosAlloc { k.removeUnusedArgs(this, args); }
case CompAlloc { m.removeUnusedArgs(this, args); }
```

2.7 Simple Flow Analysis and Optimization

Flow analysis works by using sets of *facts* to capture information about the values of variables at each point in the program. It is referred to as *flow* analysis because the calculation of fact sets is determined, not just by the individual statements in a program, but also by the control flow that links them together. The analysis that we present here is fairly simple in that it mostly focusses on the straight line code sequences in individual blocks and not on higher-level aspects of the program control flow graph such as join points, split points, and loops, as might be expected in a more full-featured data flow analysis. Nevertheless, even though the anaysis is quite simple, the information that it produces can still be used to implement some important optimizations that can significantly clean up the quality of generated MIL code.

In the following, we will represent individual facts by expressions of the form `v = t` where `v` is a program variable and `t` is a pure `Tail` expression. The restriction to pure tails (which includes allocators and pure primitives) simplifies the calculation of fact sets by avoiding the need to consider side effects. For example, if the fact `v = t` is known at the start of a code sequence `(v1 <- t1; c)`, then we can be sure that the same fact will also hold at the start of `c`, even if `t1` is not pure, provided that `v` and `v1` are distinct and that `v1` does not appear in `t`. On the other hand, if either of these latter conditions holds, then we say that the `v1 <- t1` binding *kills* the fact `v = t` as a result of assigning a (potentially) new value for `v1`.

As a simple illustration of how the results of flow analysis can be used in program optimization, suppose that we have established the fact that `v = Cons(x,y)` at the start of the following code sequence:

```
case v of
  Cons(p,q) -> b1(p,q,r)
  Nil       -> b2(s)
```

Clearly, there is actually no need for a `case` construct here because the given

fact already provides all of the information that we need to know which branch will be taken. As a result, we can replace the `case` test with a direct jump to `b1(x,y,r)` using the values for `p` and `q` that were also provided by the fact. In the following, we will refer to transformations of this kind as 'shorting out a match' because they eliminate the need for a pattern matching operation, reducing the size of the MIL code, and likely improving its runtime performance in the process. As an added benefit, we might later discover that the original binding `v <- Cons(x,y)` that established the fact `v = Cons(x,y)` is no longer required once the match on `v` has been shorted. In that case, the binding for `v` can be eliminated as dead code, resulting in further improvements in code size and run time performance, as well as reducing the total amount of heap allocation.

Another example—similar in nature because it corresponds to pairing an 'introduction' form or allocator with a subsequenct 'elimination' form or use—occurs if we encounter a `Tail` expression of the form `f @ a` at a program point where the fact `f = k{a1,...,an}` is known to hold. In general, the notation `f @ a` represents a call to an unknown function `f`. But in this particular situation, we can see that `f` is not actually unknown, and that it is in fact given by a closure, constructed using `k`. Assuming that we have access to all of the code, it is reasonable to assume that there is an associated definition for `k` of the form `k{u1,...,un} u = t` elsewhere in the program for some tail `t`, and hence we can replace the original `f @ a` expression with `[a1/u1,...,an/un,a/u]t`, using the substitution shown to instantiate each of the formal parameters with the appropriate actual values. This particular transformation eliminates the overhead of calling an unknown function via a closure but also, in much the same way as the previous example, may also turn the original closure allocation in to dead code. (Not surprisingly, there is an analogous transformation that can be applied when a fact of the form `v = b[a1,...,an]` representing the construction of a thunk reaches a tail expression of the form `invoke v`.)

Flow analysis is useful not only when dealing with allocated data structures, but also when working with operations on primitive types. As one simple example, if `neg` is the primitive unary negation operation, and if we encounter an expression of the form `neg((x))` at a point where the fact `x = neg((y))` is known to hold, then we can replace the `neg((x))` call with `return y`. (This assumes, of course, that the implementation of the `neg` primitive satisfies the law `neg((neg((y)))) = y` for all values of `y`.)

2.7.1 Facts

```
/** Lists of Facts are used to represent sets of "facts", each of
 *  which is a pair (v = t) indicating that the variable v has most
 *  recently been bound by the specified tail t (which should be either
 *  an allocator or a pure primitive call).  We can use lists of
 *  facts like this to perform dataflow analysis and optimizations on
 *  Code sequences.
 */
```

```
public class Facts(private Var v, private Tail t, private Facts next) {
  /** Remove any facts that are killed as a result of binding the variable v.
   *  The returned list will be the same as the input list vs if, and only if
   *  there are no changes to the list of facts.  In particular, this implies
   *  that we will not use any destructive updates, but it also allows us to
   *  avoid unnecessarily reallocating copies of the same list when there are
   *  no changes, which we expect to be the common case.
   */
  public static Facts kills(Var v, Facts facts) {
      if (facts!=null) {
          Facts fs = Facts.kills(v, facts.next);
          // A binding for the variable v kills any fact (w = t) that mentions v.
          if (facts.v==v || facts.t.contains(v)) {
              // Head item in facts is killed by binding v, so do not include
              // it in the return result:
              return (facts==fs) ? facts.next : fs;
          } else if (fs!=facts) {
              // Some items in facts.next were killed, but the head item
              // in facts is not, so we create a new list that retains the
              // head fact together with the facts left in fs:
              return new Facts(facts.v, facts.t, fs);
          }
      }
      return facts;
  }
}

/** Add a fact v = t to the specified list; a fact like this cannot be
 *  included if v appears in t.  (I'm not expecting the latter condition
 *  to occur often/ever, but we should be careful, just in case.)
 */
public Facts addFact(Var v, Facts facts)
  case Tail {
    return (isPure() && !contains(v)) ? new Facts(v, this, facts) : facts;
  }

/** Look for a fact about a specific variable.
 */
public static Tail lookupFact(Var v, Facts facts)
  case Facts {
    for (; facts!=null; facts=facts.next) {
      if (facts.v==v) {
        return facts.t;
      }
    }
    return null;
  }
public Tail lookupFact(Facts facts)
  case Atom { return null; }
  case Var  { return Facts.lookupFact(this, facts); }
  case Top  { return tl.lookupFact(this); /* top level can't use local facts */ }
public Tail lookupFact(Top top)
  case TopLevel   { return tail.lookupFact(top); }
  case Tail       { return null; }
  case Allocator  {
                    this.top = top; return this; }

class Allocator {
  protected Top top = null;
  public getter top;
}

/** Look for a previous computation of the specified tail in the current
 *  set of facts; note that the set of facts should only contain pure
 *  computations (allocators and pure primitive calls)..
 */
public static Var find(Tail t, Facts facts)
  case Facts {
```

```
    for (; facts!=null; facts=facts.next) {
      if (facts.t.sameTail(t)) {
        return facts.v;
      }
    }
    return null;
  }
```

2.7.2 Main Flow Analysis

```
class MILProgram {
  /** Run a flow pass over this program.  Assumes a previous call
   *  to shake() to compute call graph information, and calls inlining()
   *  automatically at the end of the flow pass, which in turn will call
   *  shake to eliminate dead code and compute updated call graph
   *  information.
SUSPECT COMMENT OUT OF DATE
   */
  public void flow() {
      for (DefnSCCs dsccs = sccs; dsccs!=null; dsccs=dsccs.next) {
          for (Defns ds=dsccs.head.getBindings(); ds!=null; ds=ds.next) {
              ds.head.flow();
          }
      }
  }
}

public void flow()
  case Defn abstract;
  case Block {
      code = code.flow(null, null);
      code.liveness();
  }
  case ClosureDefn {
      tail.liveness(null);
  }
  case TopLevel {
      // TODO: Do something here ... ?
  }

/** Optimize a Code block using a simple flow analysis.
 */
public Code flow(Facts facts, AtomSubst s)
  case Code  abstract;
  case Done  {
     t = t.apply(s);
     Code nc = t.rewrite(facts);
     return (nc==null) ? this : nc.flow(facts, s);
  }
  case Bind {
    t = t.apply(s);                 // Update tail to reflect substitution
    s = AtomSubst.remove(v, s);     // Update substitution

    // Common subexpression elimination:
    // TODO: do we need to limit the places where this is used?
    Var p = Facts.find(t, facts);   // Look for previously computed value
    if (p!=null) {
      MILProgram.report("cse: using previously computed value "+p+" for "+v);
      return c.flow(facts, new AtomSubst(v, p, s));
    }

    // Apply left monad law: (v <- return a; c) == [a/v]c
    Atom a  = t.returnsAtom();      // Check for v <- return a; c
    if (a!=null) {
      MILProgram.report("applied left monad law for "+v+" <- return "+a);
```

```
      return c.flow(facts, new AtomSubst(v, a, s));
    }

    // Look for opportunities to rewrite this tail, perhaps using previous results
    Code nc = t.rewrite(facts); // Look for ways to rewrite the tail
    if (nc!=null) {
      return nc.andThen(v, c).flow(facts, s);
    }

    // Propagate analysis to the following code, updating facts as necessary.
    c = c.flow(t.addFact(v, Facts.kills(v, facts)), s);
    return this;
  }
  case Match { // case a of alts; d
    // Look for an opportunity to short this Match
    a = a.apply(s);
    Tail t = a.lookupFact(facts);
    if (t!=null) {
      BlockCall bc = t.shortMatch(s, alts, def);
      if (bc!=null) {
          // We can't change the overall structure of the Code object that we're
          // traversing inside flow(), but we can modify this Match to eliminate
          // the alternatives and replace the default with the direct BlockCall.
          // TODO: is this comment valid?  It might have been written at a time
          // when flow was returning a set of variables, instead of a rewritten
          // Code object, because it is now possible to do a rewrite ...
/*
          return new Done(bc.rewriteBlockCall(facts));
*/
          alts = new TAlt[0];
          def  = bc.rewriteBlockCall(facts);
          return this;
      } else {
          // If a = bnot((n)), then a match of the form case a of alts can be
          // rewritten as case n of alts', where alts' flips the cases for true
          // and false in the original alternatives, alts.
          Atom n = t.isBnot();
          if (n!=null) {
            a    = n;           // match on the parameter n
            alts = new TAlt[] { // swap alternatives for True and False
                     new TAlt(Cfun.True, Var.noVars, DataAlloc.False.shortMatch(s, alts, def)),
                     new TAlt(Cfun.False, Var.noVars, DataAlloc.True.shortMatch(s, alts, def))
                   };
            def  = null;       // Remove default branch
          }
      }
      // TODO: when we continue on to the following code, possibly after applying the substitution
      // s in the cases above, might we then apply it again in what follows?  (is that a problem?)
    }

    // Match will be preserved, but we still need to update using substitution s
    // and to compute the appropriate set of live variables.
    if (def!=null) {                    // update the default branch
      def = def.applyBlockCall(s).rewriteBlockCall(facts);
    }
    Var v = (Var)a;                     // TODO: eliminate this ugly cast!
    // We do not need to kill facts about v here because we are not changing its value
    // facts = Facts.kills(v, facts);
    for (int i=0; i<alts.length; i++) { // update regular branches
      alts[i].flow(v, facts, s);
    }
    return this;
  }

public void flow(Var v, Facts facts, AtomSubst s)
  case TAlt {  // c args -> bc
    // Extends substitution with fresh bindings for the variables in
    // args.  Simple, but often unnecessary allocation.
```

```
    Var[] vs = Temp.makeTemps(args.length);
    s        = AtomSubst.extend(args, vs, s);
    // facts will be valid for bc because we've renamed using fresh vars
    facts    = new Facts(v, new DataAlloc(c).withArgs(vs), facts);
    bc       = bc.forceApplyBlockCall(s).rewriteBlockCall(facts);
    args     = vs;
  }

public Code andThen(Var v, Code rest)
  case Code  abstract;
  case Bind  { c = c.andThen(v, rest); return this; }
  case Done  { return new Bind(v, t, rest); }
  case Match {
    debug.Internal.error("append requires straightline code this code object");
    return this;
  }

Atom returnsAtom()
  case Tail   { return null; }
  case Return { return a; }

Atom isBnot()
  case Tail     { return null; }
  case PrimCall { return p==Prim.bnot ? args[0] : null; }

public Code rewrite(Facts facts)
  case Tail  { return null; }  // default behavior when no other case applies
  case Enter {
    ClosAlloc clos = f.lookForClosAlloc(facts); // Is f a known closure?
    if (clos!=null) {
      MILProgram.report("rewriting "+f+" @ "+a);
      // TODO: Is this rewrite always a good idea? Are there cases where it
      // ends up creating more work than it saves?
      Tail t = clos.withArg(a);                 // If so, apply it in place
      Code c = t.rewrite(facts);
      return (c==null) ? new Done(t) : c;
    }
    return null;
  }
  case Invoke {
    CompAlloc comp = a.lookForCompAlloc(facts); // Is f a known thunk?
    if (comp!=null) {
      MILProgram.report("rewriting invoke "+a);
      // Is this rewrite always a good idea? For example, if we have
      //    t <- m[x,y,z]; u <- invoke t; ... code that uses t ...
      // then we can rewrite to
      //    t <- m[x,y,z]; u <- m(x,y,z); ... code that uses t ...
      // but we won't be able to drop the construction of the thunk t
      // as dead code.
      Tail t = comp.invoke();                  // If so, call it directly
      Code c = t.rewrite(facts);
      return (c==null) ? new Done(t) : c;
    }
    Tail t = a.invokesTopLevel();
    return (t==null) ? null : new Done(t);
  }
  case BlockCall {
    BlockCall bc = rewriteBlockCall(facts);
    return (bc==this) ? null : new Done(bc);  // TODO: worried about this == test
  }
  case PrimCall {
    return this.rewritePrimCall(facts);
  }
  case CompAlloc {
    Allocator[] allocs = this.collectAllocs(facts);
    if (allocs!=null) {
      CompAlloc ca = deriveWithKnownCons(allocs);
      if (ca!=null) {
```

```
        MILProgram.report("deriving specialized block for CompAlloc block " + m.getId());
        return new Done(ca);
      }
    }
    return null;
  }

BlockCall rewriteBlockCall(Facts facts)
  case BlockCall {
    // Look for an opportunity to short out a Match if this block branches
    // to a Match for a variable that has a known DataAlloc value in the
    // current set of facts.
    BlockCall bc = this.shortMatch(facts);
    bc = (bc==null) ? this : bc.inlineBlockCall();

    // Look for an opportunity to specialize on known constructors:
    Allocator[] allocs = bc.collectAllocs(facts); // null;
    this.allox = allocs; // TODO: temporary, for inspection of results.
    if (allocs!=null) {
      BlockCall bc1 = bc.deriveWithKnownCons(allocs);
      if (bc1!=null) {
        bc = bc1;
        MILProgram.report("deriving specialized block for BlockCall to block " + b.getId());
      }
    }

    // when we find, b16(t109, id, t110){-, k35{}, -}, it isn't
    // necessarily a good idea to create a specialized block if the
    // id <- k35{} line appears in this block (i.e., if id is local,
    // not a top level value) and id is used elsewhere in the block.

    return bc;
  }

class Call {
  Allocator[] collectAllocs(Facts facts) {
    int l              = args.length;
    Allocator[] allocs = null;
    for (int i=0; i<l; i++) {
      Tail t = args[i].lookupFact(facts);
      if (t!=null) {
        Allocator alloc = t.isAllocator(); // we're only going to keep info about Allocators
        if (alloc!=null) {
          if (allocs==null) {
            allocs = new Allocator[l];
          }
          allocs[i] = alloc;
        }
      }
    }
    return allocs;
  }
}
```

One final situation where `deriveWithInvoke()` is useful is in the handling of `Tail` expressions of the form `invoke tl`, where `tl` has been defined at the top-level by a definition of the form `(tl <- bc)` for some block call `bc`. In this case, we expect that every call to `bc` returns a thunk and that the overall effect of `invoke tl` is the same as running the corresponding derived block call, `bc'`.

```
public Tail invokesTopLevel()
  case Atom     { return null; }
  case Top      { return tl.invokesTopLevel(); }
  case TopLevel {
```

```
    MILProgram.report("replacing invoke " + getId() + " with block call");
    return this.toBlockCall().deriveWithInvoke();
  }

/** Return a BlockCall for a TopLevel value, possibly introducing a new
 *  (zero argument) block to hold the original code for the TopLevel value.
 */
public BlockCall toBlockCall()
  case TopLevel {
    BlockCall bc = tail.isBlockCall();
    if (bc==null) {
      Block b = new Block(new Done(tail));
      b.setFormals(new Var[0]);  // TODO: lift this to global constant
      bc = new BlockCall(b);
      bc.withArgs(new Atom[0]);
      tail = bc;
    }
    return bc;
  }
```

2.7.3 Liveness

```
/** Live variable analysis on a section of code; rewrites bindings v <- t using
 *  a wildcard, _ <- t, if the variable v is not used in the following code.
 */
Vars liveness()
  case Code  abstract;
  case Done  { return t.liveness(null); }
  case Bind  {
    Vars vs = c.liveness();
    // Stub out this variable with a wildcard
    if (v!=Wildcard.obj && !Vars.isIn(v,vs)) {
      MILProgram.report("liveness replaced " + v + " with a wildcard");
      v = Wildcard.obj;
    }
    // Return the set of variables that are used in (v <-t; c)
    Vars ws = Vars.remove(v,vs);  // find variables in rest of code
    vs      = t.liveness(ws);     // add variables mentioned only in t
    return vs;
  }
  case Match {
    Vars vs = (def==null) ? null : def.liveness(null);
    for (int i=0; i<alts.length; i++) {
      vs = Vars.add(alts[i].liveness(), vs);
    }
    a = a.shortTopLevel();
    return a.add(vs);
  }
  case TAlt {  // c args -> bc
    return Vars.remove(args, bc.liveness(null));
  }

/** Special case treatment for top-level bindings of the form  x <- return y;
 *  we want to short out such bindings whenever possible by replacing all
 *  occurrences of x with y.
 *  TODO: is this comment out of date?
 */
Vars liveness(Vars vs)
  case Tail abstract;
  case Return, Invoke {
    a = a.shortTopLevel();
    return a.add(vs);
  }
  case Enter {
    a = a.shortTopLevel();
```

```
    f = f.shortTopLevel();
    return f.add(a.add(vs));
  }
  case Call {
    for (int i=0; i<args.length; i++) {
      args[i] = args[i].shortTopLevel();
      vs = args[i].add(vs);
    }
    return vs;
  }

Atom shortTopLevel()
  case Atom   { return this; }
  case Top    { return tl.shortTopLevel(this); }

Atom shortTopLevel(Atom d)
  case TopLevel { return tail.shortTopLevel(d); }
  case Tail     { return d; }
  case Return   { return a; }
```

2.7.4 Tracking Allocators at a BlockCall

```
class BlockCall {
  private Allocator[] allox;
  public void display() > {
    if (allox!=null) {
      Allocator.display(allox);
    }
  }
}

class Allocator {
  public static void display(Allocator[] allocs) {
    System.out.print("{");
    for (int i=0; i<allocs.length; i++) {
      if (i>0) { System.out.print(", "); }
      if (allocs[i]==null) {
        System.out.print("-");
      } else {
        allocs[i].display();
      }
    }
    System.out.print("}");
  }
}
```

2.7.5 Entering Function Closures Directly

An Enter of the form `f @ a` with fact `f = k{x..}` where `k{x..} a = t` can be rewritten as `t`, eliminating the cost of the enter and perhaps turning the allocation of `f` into dead code.

```
/** Test to determine if this Atom is known to hold a specific function
 *  value, represented by a ClosAlloc/closure allocator, according to the
 *  given set of facts.
 */
ClosAlloc lookForClosAlloc(Facts facts)
  case Atom { return null; }
  case Top  { return tl.lookForClosAlloc(); }
  case Var  {
```

```
    Tail t = Facts.lookupFact(this, facts);
    return t==null ? null : t.lookForClosAlloc();
  }

/** Test to determine whether this Code/Tail value corresponds to a
 *  closure allocator, returning either a ClosAlloc value, or else
 *  a null result.
 */
ClosAlloc lookForClosAlloc()
  case Code, Tail { return null; }
  case ClosAlloc  { return this; }
  case TopLevel   { return tail.lookForClosAlloc(); }
  case Done       { return t.lookForClosAlloc(); }

/** Compute a Tail that gives the result of applying this ClosAlloc value
 *  to a specified argument a.
 */
Tail withArg(Atom a)
  case ClosAlloc { return k.withArgs(args, a); }

/** Compute a Tail that gives the result of entering this closure given the
 *  arguments that are stored in the closure and the extra argument that
 *  prompted us to enter this closure in the first place.
 */
Tail withArgs(Atom[] args, Atom a)
  case ClosureDefn {
    return tail.forceApply(new AtomSubst(arg, a,
                           AtomSubst.extend(stored, args, null)));
  }
```

2.7.6 Entering Monadic Thunks Directly

```
/* In a code sequence  v <- invoke w; c  with the fact w = m[x..],
 * we can rewrite the program as  v <- m(x..); c  which makes the
 * call to the code in the thunk direct, and may turn the original
 * w <- m[x..] into dead code, eliminating the need for an allocation.
 */

/** Test to see whether this Atom, used as the argument of an invoke,
 *  is known to have been assigned to a monadic thunk.  Returns either
 *  the monadic thunk (a CompAlloc value) or null the Atom has some
 *  other value or if there is no known value in the set of facts.
 */
CompAlloc lookForCompAlloc(Facts facts)
  case Atom { return null; }
  case Top  { return tl.lookForCompAlloc(); }
  case Var  {
    Tail t = Facts.lookupFact(this, facts);
    return t==null ? null : t.lookForCompAlloc();
  }

/** Test to see if this Code/Tail is a CompAlloc (that is, if it has
 *  the form m[x..] for some m and x..).
 */
CompAlloc lookForCompAlloc()
  case Code, Tail { return null; }
  case CompAlloc  { return this; }
  case TopLevel   { return tail.lookForCompAlloc(); }
  case Done       { return t.lookForCompAlloc(); }

/** Compute the direct block call m(x..) for a monadic thunk m[x..].
 */
Call invoke()
  case CompAlloc { return new BlockCall(m).withArgs(args); }
```

2.7.7 Shorting Out Matches

If we find a `BlockCall`, `b(args)`, where `b(args) = case a of alts; d` and the flow information tells us that `a = C(...)` for some specific `Cfun C`, then we can "short" the Match by replacing the original `BlockCall` with the block call in the alternative corresponding to `C` (or the default, `d`, if there is no matching alternative). Shorting a match will avoid the cost of an unnecessary run-time test. If we are lucky, then it will also turn the original allocation into dead code.

```
BlockCall shortMatch(Facts facts)
  case BlockCall { return b.shortMatch(args, facts); }

/** Test to determine whether there is a way to short out a Match
 *  from a call to this block with the specified arguments, and
 *  given the set of facts that have been computed.  We start by
 *  querying the code in the Block to determine if it starts with
 *  a Match; if not, then the optimization will not apply and a null
 *  result is returned.
 */
BlockCall shortMatch(Atom[] args, Facts facts)
  case Block { return code.shortMatch(formals, args, facts); }

/** Test to see if this code is Match that can be shorted out. Even
 *  If we find a Match, we still need to check for a relevant item
 *  in the set of Facts (after applying a subsitution that
 *  captures the result of entering the block that starts with the
 *  Match).  Again, if it turns out that the optimization cannot be
 *  used, then we return null.
 */
BlockCall shortMatch(Var[] formals, Atom[] args, Facts facts)
  case Code  { return null; }
  case Match {
    AtomSubst s = AtomSubst.extend(formals, args, null);
    Tail      t = a.apply(s).lookupFact(facts);
    return (t==null) ? null : t.shortMatch(s, alts, def);
  }

/** Figure out the BlockCall that will be used in place of the original
 *  after shorting out a Match.  Note that we require a DataAlloc fact
 *  for this to be possible (closures and monadic thunks shouldn't show
 *  up here if the program is well-typed, but we'll check for this just
 *  in case).  Once we've established an appropriate DataAlloc, we can
 *  start testing each of the alternatives to find a matching constructor,
 *  falling back on the default branch if no other option is available.
 */
BlockCall shortMatch(AtomSubst s, TAlt[] alts, BlockCall d)
  case Tail { return null; }
  case DataAlloc {
    for (int i=0; i<alts.length; i++) { // search for matching alternative
      BlockCall bc = alts[i].shortMatch(s, c, args);
      if (bc!=null) {
        MILProgram.report("shorting out match on constructor "+c.getId());
        return bc;
      }
    }
    MILProgram.report("shorting out match using default for "+c.getId());
    return d.applyBlockCall(s); // use default branch if no match found
  }

/** Test to determine whether this alternative will match a data value
 *  constructed using a specified constructor and argument list.  The
 *  substitution captures the original instantiation of the block as
 *  determined by the original BlockCall.
 */
```

```
BlockCall shortMatch(AtomSubst s, Cfun c, Atom[] args)
  case TAlt {
    if (c==this.c) {   // constructors match
      return bc.applyBlockCall(AtomSubst.extend(this.args, args, s));
    }
    return null;
  }
```

2.8 Optimizations for Primitives

TODO: Add some strength reduction code, a division and mod operator, a subtract function, etc..

```
Code rewritePrimCall(Facts facts)
  case PrimCall {
    @primitiveRewrites
    return null;
  }

class PrimCall {
  static Code done(Tail t)                 { return new Done(t); }
  static Code done(Atom a)                 { return done(new Return(a)); }
  static Code done(int n)                  { return done(new Const(n)); }
  static Code done(Prim p, Atom[] args)    { return done(new PrimCall(p, args)); }
  static Code done(Prim p, Atom a)         { return done(new PrimCall(p, a)); }
  static Code done(Prim p, Atom a, Atom b) { return done(new PrimCall(p, a, b)); }
  static Code done(Prim p, Atom a, int n)  { return done(new PrimCall(p, a, n)); }
}

class PrimCall {
  /** Convenience constructor for PrimCalls.
   */
  public PrimCall(Prim p, Atom[] args) {
    this(p);
    withArgs(args);
  }

  /** Convenience constructor for unary PrimCalls.
   */
  public PrimCall(Prim p, Atom a) {
    this(p, new Atom[] {a});
  }

  /** Convenience constructor for binary PrimCalls.
   */
  public PrimCall(Prim p, Atom a, Atom b) {
    this(p, new Atom[] {a, b});
  }
  public PrimCall(Prim p, Atom a, int n) {
    this(p, a, new Const(n));
  }
}

/** Test to see if this tail expression is a call to a specific primitive,
 *  returning null in the (most likely) case that it is not.
 */
Atom[] isPrim(Prim p)
  case Tail     { return null; }
  case PrimCall { return (p==this.p) ? args : null; }
```

2.8.1 Rewriting for Unary Primitives

```
macro UnaryRewrite(P) {
  class PrimCall {
    @primitiveRewrites > {
      if (p==Prim.P) {
        Atom  x = args[0];
        Const a = x.isConst();
        return (a==null) ? P\Var(x, facts) : P\Const(a.getVal());
      }
    }
  }
}

macro UnaryRewrite(bnot)
class PrimCall {
  Code bnotVar(Atom x, Facts facts) {
    Tail a = x.lookupFact(facts);
    if (a!=null) {

      // Eliminate double negation:
      Atom[] ap = a.isPrim(Prim.bnot);
      if (ap!=null) {
        MILProgram.report("eliminated double bnot");
        return done(ap[0]); // bnot(bnot(u)) == u
      }

      // Handle negations of relational operators:
      if ((ap = a.isPrim(Prim.eq))!=null) {  //  eq --> neq
        MILProgram.report("replaced bnot(eq(x,y)) with neq(x,y)");
        return done(Prim.neq, ap);
      }
      if ((ap = a.isPrim(Prim.neq))!=null) {  //  neq --> eq
        MILProgram.report("replaced bnot(neq(x,y)) with eq(x,y)");
        return done(Prim.eq, ap);
      }
      if ((ap = a.isPrim(Prim.lt))!=null) {   //  lt --> gte
        MILProgram.report("replaced bnot(lt(x,y)) with gte(x,y)");
        return done(Prim.gte, ap);
      }
      if ((ap = a.isPrim(Prim.lte))!=null) {  //  lte --> gt
        MILProgram.report("replaced bnot(lte(x,y)) with gt(x,y)");
        return done(Prim.gt, ap);
      }
      if ((ap = a.isPrim(Prim.gt))!=null) {   //  gt --> lte
        MILProgram.report("replaced bnot(gt(x,y)) with lte(x,y)");
        return done(Prim.lte, ap);
      }
      if ((ap = a.isPrim(Prim.gte))!=null) {  //  gte --> lt
        MILProgram.report("replaced bnot(gte(x,y)) with lt(x,y)");
        return done(Prim.lt, ap);
      }
    }
    return null;
  }
  static Code bnotConst(int n) {
    // No opportunity for constant folding here because bnot
    // takes a boolean (True() or False()) and not an int arg.
    return null;
  }
}

macro UnaryRewrite(not)
class PrimCall {
  Code notVar(Atom x, Facts facts) {
    Tail a = x.lookupFact(facts);
    if (a!=null) {
```

```
      // Eliminate double negation:
      Atom[] ap = a.isPrim(Prim.not);
      if (ap!=null) {
        MILProgram.report("eliminated double not");
        return done(ap[0]); // not(not(u)) == u
      }
    }
    return null;
  }
  static Code notConst(int n) {
    MILProgram.report("constant folding for not");
    return done(~n);
  }
}

macro UnaryRewrite(neg)
class PrimCall {
  Code negVar(Atom x, Facts facts) {
    Tail a = x.lookupFact(facts);
    if (a!=null) {
      Atom[] ap = a.isPrim(Prim.neg);
      if (ap!=null) {
        MILProgram.report("rewrite: -(-x) ==> x");
        return done(ap[0]); // neg(neg(u)) == u
      }
      if ((ap = a.isPrim(Prim.sub))!=null) {
        MILProgram.report("rewrite: -(x - y) ==> y - x");
        return done(Prim.sub, ap[1], ap[0]);
      }
    }
    return null;
  }
  static Code negConst(int n) {
    MILProgram.report("constant folding for neg");
    return done(-n);
  }
}
```

2.8.2 Rewriting for Commutative Binary Primitives

```
macro CommutativeBinaryRewrite(P) {
  class PrimCall {
    @primitiveRewrites > {
      if (p==Prim.P) {
        Atom  x = args[0];
        Atom  y = args[1];
        Const a = x.isConst();
        Const b = y.isConst();
        if (a==null) {
          return (b==null) ? P\VarVar(x, y, facts)
                           : P\VarConst(x, b.getVal(), facts);
        } else if (b==null) {
          Code nc = P\VarConst(y, a.getVal(), facts);
          return (nc!=null) ? nc : done(p, y, x);
        } else {
          return P\Const(a.getVal(), b.getVal());
        }
      }
    }
  }
}

class PrimCall {
  /** Look for opportunities to simplify an expression using idempotence.
   */
```

```
static private Code idempotent(Atom x, Atom y) {
  if (x==y) { // simple idempotence
    // TODO: in an expression of the form (x & y), we could further
    // exploit idempotence if x includes an or with y (or vice versa);
    // handling this would require the addition of Prim and a Facts
    // arguments.
    MILProgram.report("rewrite: x ! x ==> x");
    return done(x);
  }
  return null;
}

/** Look for opportunities to rearrange an expression written in terms of a
 *  commutative and associative primitive p, written with an infix ! in the
 *  following code.  The assumptions are that we're trying to optimize an
 *  expression of the form (x ! y) where a and b are facts that have already
 *  been looked up for each of x and y, respectively.
 */
static private Code commuteRearrange(Prim p, Atom x, Tail a, Atom y, Tail b) {
  Atom[] ap = (a!=null) ? a.isPrim(p)      : null;
  Const  c  = (ap!=null) ? ap[1].isConst() : null;

  Atom[] bp = (b!=null) ? b.isPrim(p)      : null;
  Const  d  = (bp!=null) ? bp[1].isConst() : null;

  if (c!=null) {
    if (d!=null) {      // (u ! c) ! (w ! d)
      MILProgram.report("rewrite: (u ! c) ! (w ! d) ==> (u ! w) ! (c ! d)");
      return varVarConst(p, ap[0], bp[0], c.getVal() | d.getVal());
    } else {            // (u ! c) ! y
      MILProgram.report("rewrite: (u ! c) ! y==> (u ! y) ! c");
      return varVarConst(p, ap[0], y, c.getVal());
    }
  } else if (d!=null) { // x ! (w ! c)
    MILProgram.report("rewrite: x ! (w ! d) ==> (x ! w) ! d");
    return varVarConst(p, x, bp[0], d.getVal());
  }
  return null;
}

/** Create code for (a ! b) ! n where ! is a primitive p; a and b are
 *  variables; and n is a known constant.
 */
static private Code varVarConst(Prim p, Atom a, Atom b, int n) {
  Var v = new Temp();
  return new Bind(v, new PrimCall(p, a, b), done(p, v, n));
}

/** Generate code for a deMorgan's law rewrite.  We are trying to rewrite an expression
 *  of the form p(x, y) with an associated (possibly null) fact a for x and b for y.  If
 *  both a and b are of the form inv(_) for some specific "inverting" primitive, inv,
 *  then we can rewrite the whole formula, p(inv(u), inv(v)) as inv(q(u,v)) where q is a
 *  "dual" for p.  There are (at least) three special cases for this rule:
 *    if p=and, then q=or,  inv=not:    ~p | ~q = ~(p & q)
 *    if p=or,  then q=and, inv=not:    ~p & ~q = ~(p | q)
 *    if p=add, then q=add, inv=neg:    -p + -q = -(p + q)  (add is self-dual)
 */
static private Code deMorgan(Prim q, Prim inv, Tail a, Tail b) {
  Atom[] ap;
  Atom[] bp;
  if (a!=null && (ap = a.isPrim(inv))!=null &&
      b!=null && (bp = b.isPrim(inv))!=null) {
    MILProgram.report("applied a version of deMorgan's law");
    Var v = new Temp();
    return new Bind(v, new PrimCall(q, ap[0], bp[0]), done(inv, v));
  }
  return null;
}
```

```
}

// Arithmetic Operators: ---------------------------------------------

macro CommutativeBinaryRewrite(add)
class PrimCall {
  Code addVarVar(Atom x, Atom y, Facts facts) {
    Tail a = x.lookupFact(facts);
    Tail b = y.lookupFact(facts);
    if (a!=null || b!=null) {   // Only look for a rewrite if there are some facts
      Code nc = commuteRearrange(Prim.add, x, a, y, b);
      return ((nc!=null || (nc = deMorgan(Prim.add, Prim.neg, a, b))!=null)) ? nc : null;
    }
    return null;
  }
  Code addVarConst(Atom x, int m, Facts facts) {
    if (m==0) {                         // x + 0 == x
      MILProgram.report("rewrite: x + 0 ==> x");
      return done(x);
    }
    Tail a = x.lookupFact(facts);
    if (a!=null) {
      Atom[] ap;
      if ((ap = a.isPrim(Prim.add))!=null) {
        Const b = ap[1].isConst();
        if (b!=null) {                  // (u + n) + m == u + (m + n)
          return done(p, ap[0], b.getVal() + m);
        }
      }
    }
    return null;
  }
  static Code addConst(int n, int m) {
    MILProgram.report("constant folding for add");
    return done(n + m);
  }
}

macro CommutativeBinaryRewrite(mul)
class PrimCall {
  Code mulVarVar(Atom x, Atom y, Facts facts) {
    return commuteRearrange(Prim.mul, x, x.lookupFact(facts), y, y.lookupFact(facts));
  }
  Code mulVarConst(Atom x, int m, Facts facts) {
    if (m==0) {                         // x * 0 == 0
      MILProgram.report("rewrite: x * 0 ==> 0");
      return done(0);
    }
    if (m==1) {                         // x * 1 == x
      MILProgram.report("rewrite: x * 1 ==> x");
      return done(x);
    }
    if (m==(-1)) {                      // x * -1 == neg(x)
      MILProgram.report("rewrite: x * (-1) ==> -x");
      return done(Prim.neg, x);
    }
    Tail a = x.lookupFact(facts);
    if (a!=null) {
      Atom[] ap;
      if ((ap = a.isPrim(Prim.mul))!=null) {
        Const b = ap[1].isConst();
        if (b!=null) {                  // (u * c) * m == u * (c * m)
          return done(Prim.mul, ap[0], b.getVal() * m);
        }
      } else if ((ap = a.isPrim(Prim.add))!=null) {
        Const b = ap[1].isConst();
        if (b!=null) {                  // (u + n) * m == (u * m) + (n * m)
          Var v = new Temp();
```

```
          return new Bind(v,
                          new PrimCall(Prim.mul, ap[0], new Const(m)),
                          done(Prim.add, v, b.getVal()*m));
        }
      }
    }
    return null;
  }
  static Code mulConst(int n, int m) {
    MILProgram.report("constant folding for mul");
    return done(n * m);
  }
}

// Logical Operators: ------------------------------------------------
// Among the laws that we're not currently using:
// - absorption properties
// - stronger forms of idempotence
// - excluded middle
// - properties of xor
```

The rules for rewriting `or` and `and` primitives are very similar,but use different versions of the distributivity rule. For an outermost `and`, we use the following simple rule (with variable `u` and constants `c` and `m`):

```
(u | c) & m ==> (u & m) | (c & m)
```

For an outermost `o`, however, we limit the use of distributivity to a more complex left-hand pattern (with variable `u` and three constants, `d`, `c`, and `m`):

```
((u | d) & c) | m ==> (u & c) | ((d & c) | m)
```

These choices reflect a decision to standardize on expressions of the form `(u & c) | d` when there is a combination of an `and` and an `or` with constant arguments.

```
macro CommutativeBinaryRewrite(or)
class PrimCall {
  Code orVarVar(Atom x, Atom y, Facts facts) {
    Code nc = idempotent(x, y);
    if (nc==null) {
      Tail a = x.lookupFact(facts);
      Tail b = y.lookupFact(facts);
      if ((a!=null || b!=null) &&
          (nc = commuteRearrange(Prim.or, x, a, y, b))==null) {
        nc = deMorgan(Prim.and, Prim.not, a, b);
      }
    }
    return nc;
  }

  Code orVarConst(Atom x, int m, Facts facts) {
    if (m==0) {
      MILProgram.report("rewrite: x | 0 ==> x");
      return done(x);
    }
    if (m==(~0)) {
      MILProgram.report("rewrite: x | (~0) ==> (~0)");
      return done(~0);
    }
```

```
    Tail a = x.lookupFact(facts);
    if (a!=null) {
      Atom[] ap;
      if ((ap = a.isPrim(Prim.or))!=null) {
        Const c = ap[1].isConst();
        if (c!=null) {
          MILProgram.report("rewrite: (u | c) | m ==> u | (c | n)");
          return done(new PrimCall(Prim.or, ap[0], c.getVal() | m));
        }
      } else if ((ap = a.isPrim(Prim.not))!=null) {
        MILProgram.report("rewrite: (~u) | m ==> ~(u & ~m)");
        Var v = new Temp();
        return new Bind(v, new PrimCall(Prim.and, ap[0], ~m),
                        done(new PrimCall(Prim.not, v)));
      } else if ((ap = a.isPrim(Prim.and))!=null) {
        Const c = ap[1].isConst();          // (_ & c) | m
        if (c!=null) {
          Tail b = ap[0].lookupFact(facts); // ((b) & c) | m
          if (b!=null) {
            Atom[] bp = b.isPrim(Prim.or);  // ((_ | _) & c) | m
            if (bp!=null) {
              Const d = bp[1].isConst();    // ((_ | d) & c) | m
              if (d!=null) {
                MILProgram.report("rewrite: ((u | d) & c) | m ==> (u & c) | ((d & c) | m)");
                Var v = new Temp();
                int n = (d.getVal() & c.getVal()) | m;
                return new Bind(v, new PrimCall(Prim.and, bp[0], c),
                                done(new PrimCall(Prim.or, v, n)));
              }
            }
          }
        }
      }
    }
    return null;
  }
  static Code orConst(int n, int m) {
    MILProgram.report("constant folding for or");
    return done(n | m);
  }
}

macro CommutativeBinaryRewrite(and)
class PrimCall {
  Code andVarVar(Atom x, Atom y, Facts facts) {
    Code nc = idempotent(x, y);
    if (nc==null) {
      Tail a = x.lookupFact(facts);
      Tail b = y.lookupFact(facts);
      if ((a!=null || b!=null) &&
          (nc = commuteRearrange(Prim.and, x, a, y, b))==null) {
        nc = deMorgan(Prim.or, Prim.not, a, b);
      }
    }
    return nc;
  }
  Code andVarConst(Atom x, int m, Facts facts) {
    if (m==0) {
      MILProgram.report("rewrite: x & 0 ==> 0");
      return done(0);
    }
    if (m==(~0)) {
      MILProgram.report("rewrite: x & (~0) ==> x");
      return done(x);
    }
    Tail a = x.lookupFact(facts);           // (a) & m
    if (a!=null) {
      Atom[] ap;
```

```
      if ((ap = a.isPrim(Prim.and))!=null) {
        Const c = ap[1].isConst();
        if (c!=null) {
          MILProgram.report("rewrite: (u & c) & m ==> u & (c & n)");
          return done(new PrimCall(Prim.and, ap[0], c.getVal() & m));
        }
      } else if ((ap = a.isPrim(Prim.not))!=null) {
        MILProgram.report("rewrite: (~u) & m ==> ~(u | ~m)");
        Var v = new Temp();
        return new Bind(v, new PrimCall(Prim.or, ap[0], ~m),
                        done(new PrimCall(Prim.not, v)));
      } else if ((ap = a.isPrim(Prim.or))!=null) {
        Const c = ap[1].isConst();          // (_ | c) & m
        if (c!=null) {
          MILProgram.report("rewrite: (a | c) & m ==> (a & m) | (c & m)");
          Var v = new Temp();
          return new Bind(v, new PrimCall(Prim.and, ap[0], m),
                          done(new PrimCall(Prim.or, v, c.getVal() & m)));
        }
      }
    }
    return null;
  }
  static Code andConst(int n, int m) {
    MILProgram.report("constant folding for and");
    return done(n & m);
  }
}

// TODO: add more rewrites for xor?
macro CommutativeBinaryRewrite(xor)
class PrimCall {
  Code xorVarVar(Atom x, Atom y, Facts facts) {
    if (x==y) { // simple anihilator
      // TODO: in an expression of the form (x ^ y), we could further
      // exploit anihilation if x includes an or with y (or vice versa).
      MILProgram.report("rewrite: x ^ x ==> 0");
      return done(0);
    }
    return commuteRearrange(Prim.mul, x, x.lookupFact(facts), y, y.lookupFact(facts));
  }
  Code xorVarConst(Atom x, int m, Facts facts) {
    if (m==0) {                         // x ^ 0 == x
      MILProgram.report("rewrite: x ^ 0 ==> x");
      return done(x);
    }
    if (m==(~0)) {                      // x ^ (~0) == not(x)
      MILProgram.report("rewrite: x ^ (~0) ==> not(x)");
      return done(new PrimCall(Prim.not, x));
    }
    return null;
  }
  static Code xorConst(int n, int m) {
    MILProgram.report("constant folding for xor");
    return done(n ^ m);
  }
}
```

2.8.3 Rewriting for Noncommutative Binary Primitives

```
macro NoncommutativeBinaryRewrite(P) {
  class PrimCall {
    @primitiveRewrites > {
      if (p==Prim.P) {
        Atom  x = args[0];
```

```
        Atom  y = args[1];
        Const a = x.isConst();
        Const b = y.isConst();
        return (a==null)
                ? ((b==null) ? P\VarVar(x, y, facts)
                             : P\VarConst(x, b.getVal(), facts))
                : ((b==null) ? P\ConstVar(a.getVal(), y, facts)
                             : P\Const(a.getVal(), b.getVal()));
      }
    }
  }
}

// TODO: do more with rewrites for sub ...
macro NoncommutativeBinaryRewrite(sub)
class PrimCall {
  Code subVarVar(Atom x, Atom y, Facts facts) {
    if (x==y) {                       // x - x == 0
      MILProgram.report("rewrite: x - x ==> 0");
      return done(0);
    }
    return null;
  }
  Code subVarConst(Atom x, int m, Facts facts) {
    if (m==0) {                       // x - 0 == x
      MILProgram.report("rewrite: x - 0 ==> x");
      return done(x);
    }
    // TODO: not sure about this one; it turns simple decrements like sub(x,1) into
    // adds like add(x, -1); I guess this could be addressed by a code generator that
    // doesn't just naively turn every add(x,n) into an add instruction ...
    //
    // If n==0,  then add(x,n) shouldn't occur ...
    //    n==1,  then add(x,n) becomes an increment instruction
    //    n>1,   then add(x,n) becomes an add with immediate argument
    //    n==-1, then add(x,n) becomes a decrement instruction
    //    n< -1, then add(x,n) becomes a subtract with immediate argument
    //
    return done(Prim.add, x, (-m));     // x - n == x + (-n)
  }
  Code subConstVar(int n, Atom y, Facts facts) {
    if (n==0) {                       // 0 - y == -y
      MILProgram.report("rewrite: 0 - y ==> -y");
      return done(Prim.neg, y);
    }
    return null;
  }
  static Code subConst(int n, int m) {
    MILProgram.report("constant folding for sub");
    return done(n - m);
  }
}

macro NoncommutativeBinaryRewrite(shl)
class PrimCall {
  Code shlVarVar(Atom x, Atom y, Facts facts) {
    return null;
  }
  Code shlVarConst(Atom x, int m, Facts facts) {
    if (m==0) {                       // x << 0 == x
      MILProgram.report("rewrite: x << 0 ==> x");
      return done(x);
    }
    return null;
  }
  Code shlConstVar(int n, Atom y, Facts facts) {
    if (n==0) {                       // 0 << y == 0
      MILProgram.report("rewrite: 0 << y ==> 0");
```

```
      return done(0);
    }
    return null;
  }
  static Code shlConst(int n, int m) {
    MILProgram.report("constant folding for shl");
    return done(n << m);
  }
}

macro NoncommutativeBinaryRewrite(shr)
class PrimCall {
  Code shrVarVar(Atom x, Atom y, Facts facts) {
    return null;
  }
  Code shrVarConst(Atom x, int m, Facts facts) {
    if (m==0) {                          // x >> 0 == x
      MILProgram.report("rewrite: x >> 0 ==> x");
      return done(x);
    }
    return null;
  }
  Code shrConstVar(int n, Atom y, Facts facts) {
    if (n==0) {                          // 0 >> y == 0
      MILProgram.report("rewrite: 0 >> y ==> 0");
      return done(0);
    }
    return null;
  }
  static Code shrConst(int n, int m) {
    MILProgram.report("constant folding for shr");
    return done(n >> m);
  }
}
```

2.8.4 Constant folding for relational operators

Should this be expanded to support other tests beyond just contant folding? For example, what about simplifying `(x+m)<=(y+n)`, etc.?

```
macro RelationalBinaryRewrite(P) {
  class PrimCall {
    @primitiveRewrites > {
      if (p==Prim.P) {
        Atom  x = args[0];
        Atom  y = args[1];
        Const a = x.isConst();
        if (a!=null) {
            Const b = y.isConst();
            if (b!=null) {
                return P\Const(a.getVal(), b.getVal());
            }
        }
        return null;
      }
    }
  }
}

class DataAlloc {
  /** Special constructor for known constructor and argument list.
   */
  private DataAlloc(Cfun c, Atom[] args) {
      this.c    = c;
```

```
      this.args = args;
  }

  public static final DataAlloc True  = new DataAlloc(Cfun.True,  Var.noVars);
  public static final DataAlloc False = new DataAlloc(Cfun.False, Var.noVars);
}

class PrimCall {
    static private Code toBool(boolean b) {
        return new Done(b ? DataAlloc.True : DataAlloc.False);
//        DataAlloc d = new DataAlloc(b ? Cfun.True : Cfun.False);
//        return new Done(d.withArgs(new Atom[0]));
    }
}

macro RelationalBinaryRewrite(eq)
class PrimCall {
  static Code eqConst(int n, int m) {
    MILProgram.report("constant folding for eq");
    return toBool(n == m);
  }
}

macro RelationalBinaryRewrite(neq)
class PrimCall {
  static Code neqConst(int n, int m) {
    MILProgram.report("constant folding for neq");
    return toBool(n != m);
  }
}

macro RelationalBinaryRewrite(lt)
class PrimCall {
  static Code ltConst(int n, int m) {
    MILProgram.report("constant folding for lt");
    return toBool(n < m);
  }
}

macro RelationalBinaryRewrite(gt)
class PrimCall {
  static Code gtConst(int n, int m) {
    MILProgram.report("constant folding for gt");
    return toBool(n > m);
  }
}

macro RelationalBinaryRewrite(lte)
class PrimCall {
  static Code lteConst(int n, int m) {
    MILProgram.report("constant folding for lte");
    return toBool(n <= m);
  }
}

macro RelationalBinaryRewrite(gte)
class PrimCall {
  static Code gteConst(int n, int m) {
    MILProgram.report("constant folding for gte");
    return toBool(n >= m);
  }
}
```

2.9 Eliminating Duplicate Blocks

Note: won't work for recursive blocks.

2.9.1 Summarizing Code Sequences

This section describes an algorithm for computing an integer 'summary' of a MIL code sequence. The key property of this algorithm is that alpha equivalent code sequences have the same summary values, and hence we can quickly rule out the possibility that any particular pair of blocks might be equivalent by checking to see if their associated summaries are distinct. In effect, our algorithm for calculating summary values is much like a hash function on code sequences except that it produces a value that depends only on the overall structure of the input code sequence, and not on the choice of variables within that code. It is easy to see that the following set of definitions for the `summary()` method has the necessary property; there are many specific details in the implementation, many of which are essentially arbitrary, but it should be clear that these methods implement a pure function whose result does not depend on the variables that are used: the case for `Atom`, which covers all variables, returns a constant value.

```
/** Compute an integer summary for a fragment of MIL code with the key property
 *  that alpha equivalent program fragments have the same summary value.
 */
int summary()
  case Atom      { return -17; }
  case Const     { return val; }

  case Tail abstract;
  case Return     { return 1; }
  case Enter      { return 2; }
  case Invoke     { return 3; }
  case BlockCall  { return b.summary(args)*33; }
  case PrimCall   { return p.summary(args)*33 + 1; }
  case DataAlloc  { return c.summary(args)*33 + 2; }
  case ClosAlloc  { return k.summary(args)*33 + 3; }
  case CompAlloc  { return m.summary(args)*33 + 4; }

  case Code abstract;
  case Done       { return t.summary() * 17 + 3; }
  case Bind       { return t.summary() * 17 + c.summary() * 11 + 511; }
  case Match {
     int sum = (def==null) ? 19 : def.summary();
     if (alts!=null) {
         for (int i=0; i<alts.length; i++) {
             sum = sum*13 + alts[i].summary();
         }
     }
     return sum;
  }
  case TAlt { return 3*c.summary(args)+7*bc.summary(); }

  case Defn, Prim { return getId().hashCode(); }

/** Calculate a summary value for a list of Atom values, typically the arguments
 *  in a Call.
 */
int summary(Atom[] args)
```

```
case Block, ClosureDefn, Prim, Cfun {
    int sum = summary();
    for (int i=0; i<args.length; i++) {
        sum = 53*sum + args[i].summary();
    }
    return sum;
}
```

Also like a hash function, we would hope that the calculation of summaries uses the full range of integer results, uniformly distributed, so as to decrease the probability of a false positive (i.e., of finding inputs that are not equivalent even though they have the same summary value). Our use of multiplication by small primes in the definition of the `summary()` methods above is intended to encourage this behavior, but we have not conducted a detailed study of its effectiveness. For now, at least, it seems to work fairly well in practice.

2.9.2 Testing for Equivalent Sequences of Code

Calculation of summary values gives us a quick way to determine when two sequences of code are not equivalent. But if we find two code sequences that produce the same summary code, then a more careful comparison is needed before we conclude that the equal summary values were not just a coincidence, and that the two code sequences are indeed equivalent.

This section describes an algorithm for testing pairs of blocks, code sequences, and tails, to determine whether they are alpha equivalent. The algorithm works by maintaining a lists of the bound variables in each of the two items that are being compared. This gives us a way to determine whether the occurrence of a particular variable in the first item matches the occurrence of a (potentially) distinct variable in the second: we simply compare the positions of the two variables in the respective lists. As we descend further in to the structure of the items that are being compared, we push new entries on to the front of the respective lists every time we encounter a new bound variable. Of course, the algorithm can terminate immediately if we find any parts of the two code sequences that do not match.

In the following definitions, we typically refer to the two items that are being compared as `this` and `that`, and to the associated lists of variables as `thisvars` and `thatvars`, respectively. For example, the following code shows how we begin the comparison of two blocks, initially checking that the number of formal parameters in each case is the same, building initial `thisvars` and `thatvars` lists, and then using these lists as inputs to the `alphaCode()` method to compare the associated code sequences.

```
/** Test to see if two Block values are alpha equivalent.
 */
boolean alphaBlock(Block that)
  case Block {
      if (this.formals.length!=that.formals.length) {
```

```
        return false;
      }
      Vars thisvars = null;
      Vars thatvars = null;
      for (int i=0; i<formals.length; i++) {
         thisvars = new Vars(this.formals[i], thisvars);
         thatvars = new Vars(that.formals[i], thatvars);
      }
      return this.code.alphaCode(thisvars, that.code, thatvars);
  }
```

Comparing `Code` and `Tail` values is relatively straightforward, but requires a fair bit of boilerplate code to implement the necessary double dispatch. Fortunately, we can simplify the coding overhead a little by using some of the features of sweet. For example, the following definitions capture the initial dispatch in a comparison between two `Code` or `Tail` values and take advantage of the ability to specify multiple destinations for the same piece of code.

```
/** Test to see if two Code sequences are alpha equivalent.
 */
boolean alphaCode(Vars thisvars, Code that, Vars thatvars)
  case Code abstract;
  case Done, Bind, Match { return that.alpha@class(thatvars, this, thisvars); }

/** Test to see if two Tail expressions are the alpha equivalent.
 */
boolean alphaTail(Vars thisvars, Tail that, Vars thatvars)
  case Tail abstract;
  case Return, Enter, Invoke, BlockCall, PrimCall,
       DataAlloc, ClosAlloc, CompAlloc { return that.alpha@class(thatvars, this, thisvars); }
```

For example, if we call the `alphaCode()` method on an object of type `Bind`, then it will produce a call of the `alphaBind()` method with the second (`that`) object as its receiver. As such, our next task is to implement all of these `alphaX()` methods, each of which will require a default case in the base class that returns `false`, and then an override in the appropriate class `X` that implements the appropriate comparison between two `X` objects. The following code defines a sweet macro, `Alpha`, to capture the basic pattern, and then invokes it multiple times for each of the appropriate `Base` and `X` combinations that we need. (Note that the ! character after each macro call is necessary to ensure that sweet does not interpret the call as an attempt to redefine the `Alpha` macro.)

```
macro Alpha(Base, X) {
    /** Test two items for alpha equivalence.
     */
    boolean alpha\X(Vars thisvars, X that, Vars thatvars)
      case Base { return false; }
      case X
}

// Specific instances for alpha equivalence testing of Code values:
macro Alpha(Code,Done)  ! { return this.t.alphaTail(thisvars, that.t, thatvars); }
macro Alpha(Code,Bind)  ! {
  return this.t.alphaTail(thisvars, that.t, thatvars)
         && this.c.alphaCode(new Vars(this.v, thisvars), that.c, new Vars(that.v, thatvars));
}
macro Alpha(Code,Match) ! {
```

```
  if (!this.a.alphaAtom(thisvars, that.a, thatvars)) { // Compare discriminants:
      return false;
  }

  if (this.def==null) {                               // Compare default branches:
      if (that.def!=null) {
          return false;
      }
  } else if (that.def==null || !this.def.alphaTail(thisvars, that.def, thatvars)) {
      return false;
  }

  if (this.alts==null) {                              // Compare alternatives:
      return that.alts==null;
  } else if (that.alts==null || this.alts.length!=that.alts.length) {
      return false;
  }
  for (int i=0; i<alts.length; i++) {
      if (!this.alts[i].alphaTAlt(thisvars, that.alts[i], thatvars)) {
          return false;
      }
  }
  return true;
}

// Specific instances for alpha equivalence testing of Code values:
macro Alpha(Tail,Return)    ! { return this.a.alphaAtom(thisvars, that.a, thatvars); }
macro Alpha(Tail,Enter)     ! { return this.f.alphaAtom(thisvars, that.f, thatvars)
                                       && this.a.alphaAtom(thisvars, that.a, thatvars); }
macro Alpha(Tail,Invoke)    ! { return this.a.alphaAtom(thisvars, that.a, thatvars); }
macro Alpha(Tail,BlockCall) ! { return this.b==that.b && this.alphaArgs(thisvars, that, thatvars); }
macro Alpha(Tail,PrimCall)  ! { return this.p==that.p && this.alphaArgs(thisvars, that, thatvars); }
macro Alpha(Tail,DataAlloc) ! { return this.c==that.c && this.alphaArgs(thisvars, that, thatvars); }
macro Alpha(Tail,ClosAlloc) ! { return this.k==that.k && this.alphaArgs(thisvars, that, thatvars); }
macro Alpha(Tail,CompAlloc) ! { return this.m==that.m && this.alphaArgs(thisvars, that, thatvars); }
```

The code above references the following two auxiliaries for comparing alternatives (as part of a `Match` construct) and for comparing lists of arguments (as part of a `Call` expression):

```
/** Test to see if two alternatives are alpha equivalent.
 */
boolean alphaTAlt(Vars thisvars, TAlt that, Vars thatvars)
  case TAlt {
     if (this.c!=that.c || this.args.length!=that.args.length) {
         return false;
     }
     for (int i=0; i<args.length; i++) {
         thisvars = new Vars(this.args[i], thisvars);
         thatvars = new Vars(that.args[i], thatvars);
     }
     return this.bc.alphaBlockCall(thisvars, that.bc, thatvars);
  }

/** Test to see if two Call expressions have alpha equivalent argument lists.
 */
boolean alphaArgs(Vars thisvars, Call that, Vars thatvars)
  case Call {
      for (int i=0; i<args.length; i++) {
          if (!this.args[i].alphaAtom(thisvars, that.args[i], thatvars)) {
              return false;
          }
      }
      return true;
  }
```

The comparison of two `Atom` values uses a simple version of the double dispatch pattern that we saw previously for `Code` and `Tail` values:

```
/** Test to see if two atoms are the same upto alpha renaming.
 */
boolean alphaAtom(Vars thisvars, Atom that, Vars thatvars)
  case Atom { return this.sameAtom(that); }
  case Var  { return that.alphaVar(thatvars, this, thisvars); }

macro Alpha(Atom, Var) ! {
    int thisidx = Vars.lookup(this, thisvars);
    int thatidx = Vars.lookup(that, thatvars);
    return (thisidx==thatidx && (thisidx>=0 || this.sameAtom(that)));
}
```

Note, in particular, that the comparison of two variables is implemented as described previously by finding and comparing the variable positions in the `thisvars` and `thatvars` lists, and not by comparing the variables directly against one another.

```
class Vars {
  /** Find the position of a specific variable in the given list,
   *  or return (-1) if the variable is not found.
   */
  public static int lookup(Var v, Vars vs) {
      for (int pos = 0; vs!=null; vs=vs.next) {
          if (v==vs.head) {
              return pos;
          } else {
              pos++;
          }
      }
      return (-1);
  }
}
```

2.9.3 Identifying Duplicated Definitions

Given the `summary()` and `alphaBlock()` methods defined in the previous sections, we can now implement an algorithm for scanning all of the block definitions in a given program to calculate appropriate summaries and to look for pairs of blocks that have equivalent code. To store the results of this analysis, we will add the following two extra fields to each block:

```
class Block {
   /** Holds the most recently computed summary value for this block.
    */
   private int summary;

   /** Points to a different block with equivalent code, if one has been
    *  identified.  A null value indicates that there is no replacement
    *  block.
    */
   private Block replaceWith = null;
}
```

After we process each block, we can look for previously visited blocks with the same summary value, each of which might potentially be a duplicate of the current block. To improve the performance of our implementation, we use a hashtable-like structure to spread the collection of previously visited blocks across an array of smaller lists, and we use the value of a block's summary, modulo the length of the table, to identify the list in which we should store and search for duplicates. As we consider each block in the resulting list, we make a quick comparison of summary values and, only if there is a match, continue to perform a full `alphaBlock()` test. Eventually, we will either find a match or else reach the end of the list and can add the current block to the front of the table having established that there are no previously seen copies of the same block.

```
/** Look for a previously summarized version of this block in the table.
 *  Return true if a duplicate was found.
 */
boolean findIn(Blocks[] table)
  case Block {
     int idx = this.summary % table.length;
     if (idx<0) idx += table.length;

     for (Blocks bs = table[idx]; bs!=null; bs=bs.next) {
         if (bs.head.summary==this.summary && bs.head.alphaBlock(this)) {
             this.replaceWith = bs.head;
             MILProgram.report("Replacing " + this.getId() + " with " + bs.head.getId());
             return true;
         }
     }
     this.replaceWith = null;    // There is no replacement for this block
     table[idx]       = new Blocks(this, table[idx]);
     return false;
  }
```

The main algorithm proceeds in two phases. During the first phase, we initialize an empty `table` and then visit and compute a summary for each block, adding the appropriate entries to the table as we proceed. In the second phase, we visit each part of the abstract syntax tree a second time and eliminate references to duplicated blocks by using the information that was captured in the `replaceWith` fields during the first phase.

```
class MILProgram {
  public void collapse() {
      Blocks[] table = new Blocks[251];
      boolean found  = false;

      // Visit each block to compute summaries and populate the table:
      for (DefnSCCs dsccs = sccs; dsccs!=null; dsccs=dsccs.next) {
          for (Defns ds=dsccs.head.getBindings(); ds!=null; ds=ds.next) {
              found |= ds.head.summarizeBlocks(table);
          }
      }

      // Update the program to eliminate duplicate blocks:
      if (found) {
          for (DefnSCCs dsccs = sccs; dsccs!=null; dsccs=dsccs.next) {
              for (Defns ds=dsccs.head.getBindings(); ds!=null; ds=ds.next) {
                 ds.head.eliminateDuplicates();
```

```
            }
        }
    }
  }
}

/** Compute a summary for this definition (if it is a block) and then look for
 *  a previously encountered block with the same code in the given table.
 *  Return true if a duplicate was found.
 */
boolean summarizeBlocks(Blocks[] table)
  case Defn                  abstract;
  case Block                 { summary = code.summary(); return findIn(table); }
  case ClosureDefn, TopLevel { return false; }
```

2.9.4 Rewriting

This presentation needs to be reworked; the current presentation order (particularly the introduction of the main `collapse()` method after `summary()` and `alphaBlock()`, but before `eliminateDuplicates()`) does not make sense!

```
// TODO: How will this approach work if we end up replacing program entry points with other blocks?

// TODO: Why don't we apply the same techniques to ClosureDefns too?

void eliminateDuplicates()
  case Defn abstract;
  case Block {
      code.eliminateDuplicates();
  }
  case TopLevel, ClosureDefn {
     tail.eliminateDuplicates();
  }
  case Code abstract;
  case Done { t.eliminateDuplicates(); }
  case Bind { t.eliminateDuplicates(); c.eliminateDuplicates(); }
  case Match {
      if (alts!=null) {
          for (int i=0; i<alts.length; i++) {
              alts[i].eliminateDuplicates();
          }
      }
      if (def!=null) {
          def.eliminateDuplicates();
      }
  }
  case TAlt { bc.eliminateDuplicates(); }
  case Tail { /* nothing to do in most cases */ }
  case BlockCall { b = b.replaceWith(); }
  case CompAlloc { m = m.replaceWith(); }

Block replaceWith()
  case Block { return replaceWith==null ? this : replaceWith; }
```

2.10 Analyzing Number of Calls

```
class Defn {
  protected int numberCalls;
  protected int numberThunks;
```

```
  public getter numberCalls, numberThunks;

  public void resetCallCounts() {
    numberCalls = numberThunks = 0;
  }

  /** Register a non-tail call for this block.
   */
  public void called()  { numberCalls++;  }

  /** Register the creation of a monadic thunk for this block.
   */
  public void thunked() { numberThunks++; }
}

class MILProgram {
  public void analyzeCalls() {
    for (DefnSCCs dsccs = sccs; dsccs!=null; dsccs=dsccs.next) {
      for (Defns ds=dsccs.head.getBindings(); ds!=null; ds=ds.next) {
        ds.head.resetCallCounts();
      }
      // Now we have initialized all counts in this and preceding SCCs,
      // so it is safe to count calls in this SCC.
      for (Defns ds=dsccs.head.getBindings(); ds!=null; ds=ds.next) {
        ds.head.analyzeCalls();
      }
    }

    System.out.print("CALLED :");
    for (DefnSCCs dsccs = sccs; dsccs!=null; dsccs=dsccs.next) {
      for (Defns ds=dsccs.head.getBindings(); ds!=null; ds=ds.next) {
        int calls = ds.head.getNumberCalls();
        if (calls>0) {
          System.out.print(" " + ds.head.getId() + "[" + calls + "]");
        }
      }
    }
    System.out.println();

    System.out.print("THUNKED:");
    for (DefnSCCs dsccs = sccs; dsccs!=null; dsccs=dsccs.next) {
      for (Defns ds=dsccs.head.getBindings(); ds!=null; ds=ds.next) {
        int calls = ds.head.getNumberThunks();
        if (calls>0) {
          System.out.print(" " + ds.head.getId() + "[" + calls + "]");
        }
      }
    }
    System.out.println();
  }
}

public void analyzeCalls()
  case Defn abstract;
  case Block       { code.analyzeCalls();
 }
  case ClosureDefn { tail.analyzeCalls(); }
  case TopLevel    { tail.analyzeCalls(); }

public void analyzeCalls()
  case Code, Tail { }
  case Done       { t.analyzeTailCalls(); }
  case Bind       { t.analyzeCalls(); c.analyzeCalls(); }
  case Match      { /* a Match can only contain tail calls */ }
  case BlockCall  { b.called(); }
  case ClosAlloc  { k.thunked(); }
  case CompAlloc  { m.thunked(); }
```

```
public void analyzeTailCalls()
  case Tail       { }
  case ClosAlloc  { k.thunked(); }
  case CompAlloc  { m.thunked(); }
```

Appendix A

Library Operations

This appendix provides some general purpose data structures and algorithmic tools, each of which is packaged using one ore more sweet macros. Each of these components is used, often multiple times, in the main code but we have chosen to present them separately because they they are not in any way specific to the languages that we are working with in the rest of this document.

A.1 Singleton Classes

```
macro Singleton(Class) {
  class Class {
    private Class() {}
    public static final Class obj = new Class();
  }
}
```

A.2 Caching

```
macro Cache(Type, name, cache, arg) {
  private static Type[] cache = new Type[10];
  public static Type name(int arg) {
    if (arg>=cache.length) {
      Type[] newCache = new Type[Math.max(arg+1, 2*cache.length)];
      for (int i=0; i<cache.length; i++) {
        newCache[i] = cache[i];
      }
      cache = newCache;
    } else if (cache[arg]!=null) {
      return cache[arg];
    }
    // Code to update cache[arg] = ... will be appended here.
  }
  // A questionable use of sweet macros; we require the instantiation of this
  // macro to be followed by > { code } where code calculates and returns a
  // value to be cached in cache[arg].
```

```
  Type name(int arg)
}
```

A.3 Simple List Operations

```
/** This macro can be used to define a simple linked list of values.
 */
macro List(X) {
  public class X\s(public X head, public X\s next)
}
```

This macro can be used to place a reverse method in any class that has a basic "linked-list" structure. The first parameter specifies the name of the class, while the second is the name of the field that holds a pointer to the next list element. The reverse method generated by this code is a static and destructive implementation of list reverse.

```
macro AddReverse(Xs) {
  class Xs {
    /** Reverse a linked list of elements.
     */
    public static Xs reverse(Xs list) {
      Xs result = null;
      while (list!=null) {
        Xs temp = list.next;
        list.next   = result;
        result      = list;
        list        = temp;
      }
      return result;
    }
  }
}
```

The next macro can be used to place a length method in any class that has a basic "linked-list" structure. The first parameter specifies the name of the class, while the second is the name of the field that holds a pointer to the next list element. The length method generated by this code is static.

```
macro AddLength(Xs) {
    class Xs {
        /** Return the length of a linked list of elements.
         */
        public static int length(Xs list) {
            int len = 0;
            for (; list!=null; list=list.next) {
                len++;
            }
            return len;
        }
    }
}
```

This macro can be used to place an `isIn()` method in any class that has a basic "linked-list" structure.

```
macro AddIsIn(X) {
    class X\s {
        /** Test for membership in a list.
         */
        public static boolean isIn(X val, X\s list) {
            for (; list!=null; list=list.next) {
                if (list.head==val) {
                    return true;
                }
            }
            return false;
        }
    }
}
```

This macro can be used to place a cons method in a class that has a basic "linked-list" structure. The cons method is added to the Elem class, and builds a new List value by adding this element to the front of the next list.

```
macro AddCons(X) {
    class X {
        public X\s cons(X\s next) {
            return new X\s(this, next);
        }
    }
}
```

A.4 Ordered Lists

An ordered list is a simple linked list data structure in which each element has a associated integer key. The elements are stored in increasing key order with at most one element for any given key value. This makes it possible to implement standard set operations—including subset and membership tests as well as union, intersection, and difference—in linear time.

```
macro OrderedList(X, getN) {
  public class private X\s(private X head, private X\s next) {

    /** Determine whether x is a member of the list xs.
     */
    public static boolean member(X x, X\s xs) {
      for (int n=x.getN(); xs!=null; xs=xs.next) {
        int m = xs.head.getN();
        if (m>n)  { return false; }
        if (m==n) { return true;  }
      }
      return false;
    }

    /** Determine whether xs and ys are the same lists.
     */
    public static boolean same(X\s xs, X\s ys) {
      while (xs!=null) {
        if (ys==null || xs.head.getN()!=ys.head.getN()) {
          return false;
        }
      }
      return (ys==null);
```

```
}

/** Determine whether xs is a subset of ys.
 */
public static boolean subset(X\s xs, X\s ys) {
  if (xs==null) {
    return true;
  } else {
    // Basic idea: n is the value of the head of xs, and hence
    // the first number that we will be searching for in ys.
    for (int n=xs.head.getN(); ys!=null; ys=ys.next) {
      int m = ys.head.getN();
      if (m>n) {             // n does not appear in ys
        return false;
      } else if (m==n) {     // found a match
        if ((xs = xs.next)==null) {
          return true;       // all elements of xs were matched!
        }
        n = xs.head.getN();
      }
    }
    return false;
  }
}

/** Insert x into the list ys.
 */
public static X\s insert(X x, X\s ys) {
  X\s orig = ys;  // remember start of list
  X\s prev = null;
  int  n    = x.getN();
  while (ys!=null) {
    int m = ys.head.getN();
    if (m>n)  { break; }
    if (m==n) { return orig; }
    prev = ys;
    ys   = ys.next;
  }
  if (prev==null) {
    return new X\s(x, ys);
  } else {
    prev.next = new X\s(x, ys);
    return orig;
  }
}

/** Insert the elements of xs into the list ys.  New list nodes
 *  are created for each element of xs, even at the end of the
 *  list; we do not attempt to modify or share any portion of
 *  list structure of xs, but we are free to modify ys.
 */
public static X\s insert(X\s xs, X\s ys) {
  X\s orig = ys;  // remember start of list

  // Loop through the elements that we want to insert
  for (X\s prev=null; xs!=null; xs=xs.next) {
    int n = xs.head.getN();
    int m = 0; // to prevent uninitialized variable warning

    // Find position in ys where we should insert u.head
    for (; ys!=null && n>(m=ys.head.getN()); ys=ys.next) {
      prev = ys;
    }

    // Insert xs.head into ys at this position (if it is not already there)
    if (ys!=null && n==m) {         // Already included?
      prev = ys;
      ys   = ys.next;
```

```
    } else {                        // Need to insert ...
      X\s next = new X\s(xs.head, ys);
      prev = (prev==null) ? (orig = next) : (prev.next = next);
    }
  }
  return orig;
}

/** Remove an element x from a list ys.
 */
public static X\s remove(X x, X\s ys) {
  X\s orig = ys;  // remember start of list
  X\s prev = null;
  int  n    = x.getN();
  while (ys!=null) {
    int m = ys.head.getN();
    if (m>n) { return orig; } // element not found
    if (m==n) {
      if (prev==null) {
        return ys.next;
      } else {
        prev.next = ys.next;
        return orig;
      }
    }
    prev = ys;
    ys   = ys.next;
  }
  return orig;
}

/** Remove the elements of xs from the list ys.  Applies destructive
 *  updates to ys, but does not modify xs.
 */
public static X\s remove(X\s xs, X\s ys) {
  X\s orig = ys;  // remember start of list

  // Loop through the elements that we want to remove
  for (X\s prev=null; xs!=null; xs=xs.next) {
    int n = xs.head.getN();
    int m = 0; // to prevent uninitialized variable warning

    // Find position in ys where xs.head would occur
    for (; ys!=null && n>(m=ys.head.getN()); ys=ys.next) {
      prev = ys;
    }

    if (ys==null) {    // At end of list, element not found
      return orig;
    } else if (n==m) { // Remove a matching element
      if (prev==null) {
        orig = ys.next;
      } else {
        prev.next = ys.next;
      }
      ys = ys.next;
    }
  }
  return orig;
}

/** Intersect xs with ys, removing elements from ys that are NOT in xs.
 */
public static X\s intersect(X\s xs, X\s ys) {
  X\s orig = ys;  // remember start of list

  // Loop through the elements that we want to keep
  X\s prev = null;
```

```
      for (; xs!=null; xs=xs.next) {
        int n = xs.head.getN();
        int m = 0; // to prevent uninitialized variable warning

        // Remove elements from ys that don't match n
        for (; ys!=null && n>(m=ys.head.getN()); ys=ys.next) {
          if (prev==null) {
            orig = ys.next;
          } else {
            prev.next = ys.next;
          }
        }

        if (ys==null) {    // At end of list, element not found
          return orig;
        } else if (n==m) { // Keep a matching element
          prev = ys;
        }
        ys = ys.next;
      }
      // Drop remaining elements from ys
      if (prev==null) {
        return null;
      } else {
        prev.next = null;
        return orig;
      }
    }
  }
}

  /* Display this list of variables.
  public static void display(String msg, X\s ts) {
    System.out.print(msg + " =");
    for (; ts!=null; ts=ts.next) {
      System.out.print(" " + ts.head.getN());
    }
    System.out.println();
  }
   */
```

A.5 Dependency Analysis

The goal of a dependency analysis is to construct a directed acyclic graph of binding groups from an input list of bindings. Each binding group is just a strongly-connected component in the dependency graph, and so we will base our implementation of dependency analysis on a standard algorithm for computing the strongly-connected components of a directed graph. More specifically, we will use the algorithm described by Cormen, Leiserson and Rivest [1], which they credit to Kosaraju and Sharir. This algorithm uses two depth-first searches, one of an input graph $G = (V, E)$ and one of its transpose G^T (the graph obtained from G by reversing the direction of each edge in E), to identify the strongly connected components of G in $O(|V| + |E|)$ time. Our selection of this particular algorithm was, in fact, the primary motivation for maintaining both `callees` and `callers` information in individual `X` values. The former provides an adjacency list representation of the dependency graph, while the latter gives an adjacency list representation of its transpose.

The following sections build up an implementation of the strongly connected components algorithm, starting in Section A.5.1 with a description of the adjacency list structures that capture the structure of the input graph. During the first depth-first search, implemented by `searchForward()`, the input list of bindings is reordered so that the bindings are arranged in decreasing order of finishing time. This process is detailed in Section A.5.2. The second depth-first search, implemented by `searchReverse()`, uses this list to generate a list of `XSCC` values. The implementation of `XSCCs` is described in Section A.5.3, while the details of the actual depth-first search are presented in Section A.5.4.

A.5.1 Representing the Dependency Graph

Each `X` includes two lists of `Xs` called `callees` and `callers`, which are used to hold information about dependencies between the bindings in a given list. Every dependency `caller -> callee` will show up twice: the `caller` will be included in `callee.callers` while the `callee` will be included in `caller.callees`.

```
macro SCCLists(X) {
  class X {
    /** Records the successors/callees of this node.
     */
    private X\s callees = null;

    /** Records the predecessors/callers of this node.
     */
    private X\s callers = null;

    /** Update callees/callers information with dependencies.
     */
    public void calls(X\s xs) {
      for (callees=xs; xs!=null; xs=xs.next) {
        xs.head.callers = new X\s(this, xs.head.callers);
      }
    }
  }
}
```

A.5.2 The Forward Depth-First Search

The first depth-first search is implemented by `Xs.searchForward()`, which iterates over the `Xs` in a given list and calls `X.forwardVisit()` on each one. The latter method continues the depth-first traversal, computing reverse dependency information by calling `X.callers` at appropriate points, and making recursive calls to `X.forwardVisit()` as necessary. The result of the first depth-first search is a list of bindings arranged in reverse order of finishing times.

```
macro SCCForwardDFS(X) {
  class X\s {
    /** Depth-first search the forward dependency graph. Returns a list with the
     *  same Xs in reverse order of finishing times.  (In other words, the last
     *  node that we finish visiting will be the first node in the result list.)
     */
```

```
    public static X\s searchForward(X\s xs, X\s result) {
      for (; xs!=null; xs=xs.next) {
        result = xs.head.forwardVisit(result);
      }
      return result;
    }
  }

  class X {
    /** Flag to indicate that this node has been visited during the depth-first
     *  search of the forward dependency graph.
     */
    private boolean visited = false;

    /** Visit this X during a depth-first search of the forward dependency graph.
     */
    public X\s forwardVisit(X\s result) {
      if (!this.visited) {
        this.visited = true;
        return new X\s(this, X\s.searchForward(this.callees, result));
      }
      return result;
    }
  }
}
```

We can begin to understand how this code contributes to the calculation of strongly-connected components by noting that a call to `forwardVisit()` on a binding b that has not previously been visited will vist every binding in the same strongly connected component as `b`, as well as all of the bindings in groups that `b` depends on (unless those binding groups have already been visited). Note also that `b` will be the last of all those bindings that we finish visiting, and hence will appear before all of those bindings in the `result` list. Later, when we traverse the transpose/reverse dependency graph, we will still have the same strongly connected components, but the edges between components will be reversed. Hence the result of the second depth-first search will be a depth-first forest in which each tree corresponds directly to a strongly-connected component of the original graph.

A.5.3 Representing Binding SCCs

Before we complete the description of our dependency analysis algorithm, we will describe how values of type `BindingSCC` are used to represent individual binding groups. Of course, each `BindingSCC` is associated with a particular list of `bindings`. The `bindings` field in each `BindingSCC` is initially `null`; individual bindings can be added as the group is constructed by calling `BindingSCC.add`.

```
macro SCCs(X) {
  public class X\SCC {
    /** Records the list of X\s in this binding group.
     */
    private X\s bindings = null;

    /** Return the list of X\s in this scc.
     */
```

```
  public getter bindings;

  /** Add an X to this scc.
   */
  public void add(X binding) {
    bindings = new X\s(binding, bindings);
  }
}
```

In later stages of a compiler, it can be useful to know whether the bindings in a particular binding scc are recursive; this can be captured by including a `recursive` flag in each `X\SCC`. This flag will be initialized to `false`, but set to `true` (by calling `X\SCC.setRecursive`) if recursion is detected during the second depth-first search of our dependency analysis algorithm.

```
class X\SCC {
  /** Indicates if the bindings in this scc are recursive.  This flag is
   *  initialized to false but will be set to true if recursive bindings are
   *  discovered during dependency analysis.  If there are multiple bindings
   *  in this scc, then they must be mutually recursive (otherwise they would
   *  have been placed in different binding sccs) and this flag will be set to true.
   */
  private boolean recursive = false;

  /** This method is called when a recursive binding is
   *  discovered during dependency analysis.
   */
  public void setRecursive() {
    recursive = true;
  }

  /** Return a boolean true if this is a recursive binding scc.
   */
  public boolean isRecursive() {
    return recursive;
  }
}
```

Again, for the purposes of later stages of the compiler, we will also record dependency information between binding sccs. For simplicity, we will store only forward dependency information between binding sccs. Of course, it would be easy enough to add reverse dependency information too, if that was needed. Just as we used a type `X\s` to represent lists of `X` values, we will introduce a type `X\SCCs` to represent lists of `X\SCC` values.

```
  macro List(X\SCC)
//macro addIsIn(X\SCC)  // TODO: should use this instead of find!
  public class X\SCCs   // make the SCCs class public.  TODO: Make this neater!
```

Not surprisingly, the code that we need to record dependency information for each `X\SCC` is much like the code that we used to record similar information for `X\s` in Section A.5.1.

```
public class X\SCC {
  /** A list of the binding sccs on which this scc depends.  (This particular scc
   *  will not be included, so the graph of XSCCs will not have any cycles in it.)
```

```
     */
    private X\SCCs dependsOn = null;
    public getter dependsOn;

    /** Record a dependency between two binding sccs, avoiding
     *  self references and duplicate entries.
     */
    public static void addDependency(X\SCC from, X\SCC to) {
      if (from!=to && !find(to, from.dependsOn)) {
        from.dependsOn = new X\SCCs(to, from.dependsOn);
      }
    }

    /** Search for a specific binding scc within a given list.
     */
    public static boolean find(X\SCC scc, X\SCCs sccs) {
      for (; sccs!=null; sccs=sccs.next) {
        if (sccs.head==scc) {
            return true;
        }
      }
      return false;
    }
  }

}  // End of SCCs macro
```

A.5.4 The Reverse Depth-First Search

To complete the calculation of strongly connected components, a second depth-first search is used, this time over the reverse dependency graph. The following definition of `Xs.searchReverse` scans over the list of bindings produced by the first search. Each time it finds an unvisited binding it creates a new `XSCC`, `scc`, and then traverses the unvisited portion of the graph that can be reached from `root`. Each unvisited binding that is encountered during this traversal is added to the new `scc`. If we encounter a previously visited node with the same `scc` field, then we can be sure that this binding scc is recursive. Otherwise, we have found a previously detected binding scc that depends on this `scc` and we will update its dependency information accordingly.

```
macro SCCReverseDFS(X) {
  class X\s {
    /** Depth-first search the reverse dependency graph, using the list
     *  of bindings that was obtained in the forward search, with the
     *  latest finishers first.
     */
    public static X\SCCs searchReverse(X\s xs) {
      X\SCCs sccs = null;
      for (; xs!=null; xs=xs.next) {
        if (xs.head.getScc()==null) {
          X\SCC scc = new X\SCC();
          sccs      = new X\SCCs(scc, sccs);
          xs.head.reverseVisit(scc);
        }
      }
      return sccs;
    }
  }

  class X {
```

```
    /** Records the binding scc in which this binding has been placed.
     *  This field is initialized to null but is set to the appropriate
     *  binding scc during dependency analysis.
     */
    private X\SCC scc = null;

    /** Return the binding scc that contains this binding.
     */
    public getter scc;

    /** Visit this binding during a depth-first search of the reverse
     *  dependency graph.  The scc parameter is the binding scc in
     *  which all unvisited bindings that we find should be placed.
     */
    public void reverseVisit(X\SCC scc) {
      if (this.scc==null) {
        // If we arrive at a binding that hasn't been allocated to any SCC,
        // then we should put it in this SCC.
        this.scc = scc;
        scc.add(this);
        for (X\s callers=this.callers; callers!=null; callers=callers.next) {
          callers.head.reverseVisit(scc);
        }
      } else if (this.scc==scc) {
        // If we arrive at a binding that has the same binding scc
        // as the one we're building, then we know that it is recursive.
        scc.setRecursive();
        return;
      } else {
        // The only remaining possibility is that we've strayed outside
        // the binding scc we're building to a scc that *depends on*
        // the one we're building.  In other words, we've found a binding
        // scc dependency from this.scc to scc.
        X\SCC.addDependency(this.scc, scc);
      }
    }
  }
}
```

A.5.5 Putting it All Together

The complete algorithm is captured by the following definition:

```
macro SCC(X) {
  macro SCCLists(X)
  macro SCCForwardDFS(X)
  macro SCCs(X)
  macro SCCReverseDFS(X)

  class X\s {
    /** Calculate the strongly connected components of a list of Xs that
     *  have been augmented with dependency information.
     */
    public static X\SCCs scc(X\s xs) {
/*
      // Compute the transpose (i.e., fill in the callers fields)
      for (X\s bs=xs; bs!=null; bs=bs.next) {
        for (X\s cs=bs.head.callees; cs!=null; cs=cs.next) {
          cs.head.callers = new X\s(bs.head, cs.head.callers);
        }
      }

      debug.Log.println("Beginning SCC algorithm");
      for (X\s bs=xs; bs!=null; bs=bs.next) {
```

```
        debug.Log.print(bs.head.getId() + ": callees {");
        String punc = "";
        for (X\s cs = bs.head.callees; cs!=null; cs=cs.next) {
          debug.Log.print(punc);
          punc = ", ";
          debug.Log.print(cs.head.getId());
        }
        debug.Log.print("}, callers {");
        punc = "";
        for (X\s cs = bs.head.callers; cs!=null; cs=cs.next) {
          debug.Log.print(punc);
          punc = ", ";
          debug.Log.print(cs.head.getId());
        }
        debug.Log.println("}");
      }
*/

      // Run the two depth-first searches of the main algorithm.
      return searchReverse(searchForward(xs, null));
    }
  }
}

macro DebugSCC(X) {
  class X\SCCs {
    import debug.Log;
    public static void display(String label, X\SCCs sccs) {
      Log.println(label + ": [");
      for (; sccs!=null; sccs=sccs.next) {
        X\SCC scc = sccs.head;
        Log.print("  ");
        Log.print(scc.isRecursive() ? "rec" : "nonrec");
        Log.print(" {");
        String punc = "";
        for (X\s bs=scc.getBindings(); bs!=null; bs=bs.next) {
          Log.print(punc); punc=", ";
          Log.print(bs.head.getId());
        }
        Log.println("}");
      }
      Log.println("]");
    }
  }
}
```

A.5.6 Tracking Dependencies

```
macro Deps(X) {
  public class X\Deps {
    private X\s deps = null;

    /** Register a dependency on the specified variable.
     */
    public void addDependency(X x) {
      if (!X\s.isIn(x, deps)) {
        deps = new X\s(x, deps);
      }
    }

    public X\s extractFrom(X\s bound) {
      X\s prev = null;   // pointer to previous elem in list
      X\s exts = null;   // pointer to result list
      X\s ds   = deps;
      while (ds!=null) {
```

```
        if (X\s.isIn(ds.head, bound)) {
          X\s temp = ds.next;
          ds.next  = exts;
          exts     = ds;
          ds       = temp;
          if (prev==null) {
            deps = temp;
          } else {
            prev.next = temp;
          }
        } else {
          prev = ds;
          ds   = ds.next;
        }
      }
      return exts;
    }
  }

  // The code above requires a test for membership in X\s:
  macro AddIsIn(X)
}
```

Bibliography

[1] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.