Submitted to the 2010 ACM Symposium on Principles of Programming Languages (POPL)

Hoopl: Dataflow Optimization Made Simple

Norman Ramsey
Tufts University
nr@cs.tufts.edu

João Dias
Tufts University
dias@cs.tufts.edu

Simon Peyton Jones
Microsoft Research
simonpj@microsoft.com

Abstract

We present Hoopl, a Haskell library that makes it easy for compiler writers to implement program transformations based on dataflow analyses. The compiler writer must identify (a) logical assertions on which the transformation will be based; (b) a representation of such assertions, which should form a lattice of finite height; (c) transfer functions that approximate weakest preconditions or strongest postconditions over the assertions; and (d) rewrite functions whose soundness is justified by the assertions. Hoopl uses the algorithm of Lerner, Grove, and Chambers (2002), which can compose very simple analyses and transformations in a way that achieves the same precision as complex, handwritten "super-analyses." Hoopl will be the workhorse of a new back end for the Glasgow Haskell Compiler (version 6.12, forthcoming).

Reviewers: code examples are indexed at `http://bit.ly/jkr3K`

1. Introduction

If you write a compiler for an imperative language, you can exploit many years' work on code improvement ("optimization"). The work is typically presented as a long list of analyses and transformations, each with a different name. This presentation makes optimization appear complex and difficult. Another source of complexity is the need for synergistic combinations of optimizations; you may have to write one "super-analysis" per combination.

But optimization doesn't have to be complicated. Most optimizations work by applying well-understood techniques for reasoning about programs: assertions about states, assertions about continuations, and substitution of equals for equals. What makes optimization different from classic reasoning techniques is that in dataflow optimization, assertions are approximated, and all assertions are computed automatically.

This paper presents Hoopl (higher-order optimization library), a Haskell library that makes it easy to implement dataflow optimizations. Our contributions are as follows:

- Hoopl defines a simple interface for implementing analyses and transformations: you provide representations for assertions and for functions that transform assertions, and Hoopl computes assertions by setting up and solving recursion equations. Additional functions you provide use computed assertions to justify program transformations. Analyses and transformations built on Hoopl are small, simple, and easy to get right.
- Using the sophisticated algorithm of Lerner, Grove, and Chambers (2002), Hoopl can perform super-analyses by *interleaving* simple analyses and transformations. Interleaving is tricky to implement, but by using generalized algebraic data types and continuation-passing style, our new implementation expresses the algorithm with a clarity and a degree of static checking that has not previously been achieved.
- Hoopl helps you write correct optimizations: it statically rules out transformations that violate invariants of the control-flow graph, and dynamically it can help find the first transformation that introduces a fault in a test program (Whalley 1994).
- Hoopl's polymorphic, higher-order design makes it reusable with many languages. Hoopl is designed to help optimize imperative code with arbitrary control flow, including low-level intermediate languages and machine languages. As Benitez and Davidson (1988) have shown, all the classic scalar and loop optimizations can be performed over such codes.

We introduce dataflow optimization by analyzing and transforming example code (Section 2), thinking about and justifying classic optimizations using Hoare logic and substitution of equals for equals. To support our claim that Hoopl makes dataflow optimization easy, we explain how to create new dataflow analyses and transformations (Section 3), and we show complete implementations of significant analyses (Section 5) and transformations (Section 6) from the Glasgow Haskell Compiler. We also sketch a new implementation of interleaving (Section 7).

2. Dataflow analysis & transformation by example

In dataflow optimization, code-improving transformations are justified by assertions about programs; such assertions are often computed using strongest postconditions or weakest liberal preconditions. Typical transformations are to insert assignments to unobserved variables, to substitute equals for equals, and to remove assignments to unobserved variables. Insertion and removal can be composed to achieve "code motion." Hoopl expresses classic code improvements by composing simple transformations.

2.1 Simple transformations

Here is a sequence of assignments separated by assertions. We compute assertions by starting with the weakest assertion (`true`) and computing strongest postconditions. Variables do not alias.

```
  { true }
x = 7;
  { x == 7 }
y = 8:
  { x == 7 && y == 8 }
z = x + y;
```

In the assignment to `z`, the assertion `x == 7` justifies substituting 7 for `x`, leaving `z = 7 + y`. This transformation is traditionally called "constant propagation." We may also substitute 8 for `y`. Finally, because `7 + 8 == 15`, we may again substitute equals for equals, leaving the final assignment as

```
z = 15;
```

The final transformation, although it also substitutes equals for equals, has a different name: "constant folding."

2.2 A complex transformation

The loop optimization known as "induction-variable elimination" can be composed from simpler transformations. We begin by showing a loop that sums red pixels from an array:

```
struct pixel { double r, g, b; };
double sum_r(struct pixel a[], int n) {
  double x = 0.0;
  int i;
  for (i = 0; i < n; i++)
    x += a[i].r;
  return x;
}
```

To explain induction-variable elimination, we show the same code at the machine level, using our low-level compiler-target language, C-- (Ramsey and Peyton Jones 2000):

```
sum_r("address" bits32 a, bits32 n) {
     bits64 x; bits32 i;
     x = 0.0;
     i = 0;
 L1: if (i >= n) goto L2;
     x = %fadd(x, bits64[a+i*24]);
     i = i + 1;
     goto L1;
 L2: return x;
}
```

Induction-variable elimination replaces `i` with a new variable `p`, helping us to remove the computation `a+i*24` from the loop. Variable `p` is intended to satisfy the invariant

```
{ p == a + i * 24 }
```

Variable `i` is also used in the loop-termination test. To rewrite that test, we introduce another new variable `lim` satisfying the invariant `lim == a + n * 24`, so that `i >= n` if and only if `p >= lim`.

We implement the code improvement as a sequence of transformations. After each transformation, the observable behavior of the program is unchanged. Our first transformation declares `p` and `lim` and inserts suitable assignments. New code is boxed.

```
sum_r("address" bits32 a, bits32 n) {
     bits64 x; bits32 i; bits32 p, lim;
     x = 0.0;
     i = 0; p = a; lim = a + n * 24;
 L1: if (i >= n) goto L2;
     x = %fadd(x, bits64[a+i*24]);
     i = i + 1; p = p + 24;
     goto L1;
 L2: return x;
}
```

As written, the assignments to `p` and `lim` have no effect on the program, but they establish the assertions `p == a + i * 24` and `(i >= n) == (p >= lim)`. On the basis of these assertions, the compiler substitutes equals for equals, resulting in the new code in boxes below:

```
sum_r("address" bits32 a, bits32 n) {
     bits64 x; bits32 i; bits32 p, lim;
     x = 0.0;
     i = 0; p = a; lim = a + n * 24;
 L1: if (p >= lim) goto L2;
     x = %fadd(x, bits64[p]);
     i = i + 1; p = p + 24;
     goto L1;
 L2: return x;
}
```

Here the compiler switches from reasoning about states to reasoning about continuations. In particular, we reason about whether the value of a variable can be used by a continuation; this reasoning is called "liveness analysis." Naïve analysis would show that although `i` is not live at label `L2`, it is nevertheless live immediately after the assignment `i = i + 1` in the loop body, because the value of `i` could be used by the next iteration of the loop. But we use Lerner, Grove, and Chambers's (2002) algorithm to *interleave* liveness analysis with "dead-assignment elimination." Dead-assignment elimination removes an assignment if the variable assigned to is not live, that is, if it cannot be used by the assignment's continuation. No sequential composition of liveness analysis and dead-assignment elimination can get rid of these assignments to `i`, but interleaving analysis with transformation does the trick.[1] Interleaving (Section 7) eliminates the boxed assignments to `i`:

```
sum_r("address" bits32 a, bits32 n) {
     bits64 x; bits32 i; bits32 p, lim;
     x = 0.0;
     i = 0; p = a; lim = a + n * 24;
 L1: if (p >= lim) goto L2;
     x = %fadd(x, bits64[p]);
     i = i + 1; p = p + 24;
     goto L1;
 L2: return x;
}
```

After the insertion of assignments to `p` and `lim`, the substitution of equals for equals, and the removal of newly dead assignments to `i`, we have "eliminated the induction variable:"

```
sum_r("address" bits32 a, bits32 n) {
     bits64 x; bits32 p, lim;
     x = 0.0;
     p = a; lim = a + n * 24;
 L1: if (p >= lim) goto L2;
     x = %fadd(x, bits64[p]);
     p = p + 24;
     goto L1;
 L2: return x;
}
```

3. Making dataflow simple

The goal of dataflow optimization is to compute valid assertions, then use those assertions to justify code-improving transformations. Assertions are represented as *dataflow facts*. Dataflow facts relate to traditional program logic:

- A dataflow fact is usually equivalent to an assertion about program state or about a continuation. For example, in Section 2.1, `x == 7` is a dataflow fact that describes the program state.
- A set of dataflow facts forms a lattice. To ensure that analysis terminates, it is enough if no fact has more than finitely many distinct facts above it.

[1] You might be tempted to modify the liveness analysis so that `i = i + 1` is not considered a "use" of `i` if `i` is itself dead. This modification is tantamount to writing a single "super-analysis" that *combines* liveness analysis and dead-code elimination. In this case, writing a super-analysis is easy, but the approach does not scale: most super-analyses are more complicated than the examples shown here; the cost of writing a super-analysis does not scale linearly with the number of analyses combined; super-analyses often cannot be composed; and some super-analyses require nonstandard, handwritten traversals of the control-flow graph. Lerner, Grove, and Chambers (2002) discuss these issues in detail; Click and Cooper (1995) show both the advantages of and the programming cost of combining analyses.

Part of optimizer	*Specified by*	*Implemented by*	*How many*
Control-flow graphs	US	US	One
Nodes in a control-flow graph	YOU	YOU	Two datatypes per intermediate language
Dataflow fact F	YOU	YOU	One datatype per logic
Lattice operations	US	YOU	One set per logic
Transfer functions	US	YOU	One set per analysis
Rewrite functions	US	YOU	One set per transformation
Iterative solver functions	US	US	Two (forward & backward)
Solve-and-rewrite functions	US	US	Two (forward & backward)

Table 1. Parts of an optimizer built with Hoopl

- Each analysis or transformation may use a different lattice of dataflow facts.

An assertion about a continuation is an assertion about paths *from* a program point to the procedure exit; such assertions are established by a *backward dataflow analysis*. An assertion about paths *to* a program point from the procedure entry is established by a *forward dataflow analysis*. As an important special case, an assertion, such as `x == 7` above, may say simply that all paths to a point establish a predicate which describes the program state at that point.

A program point is represented as an edge in a *control-flow graph*. Edges connect nodes, each of which represents a label, an assignment, or a control transfer.

To write a dataflow *analysis*, you must

- Choose a representation F of dataflow facts and a logical interpretation thereof.
- Implement lattice operations over F (Section 3.1).
- Write *transfer functions* that relate dataflow facts before and after each type of node (Section 3.2).

To write a *transformation* based on an analysis, you must also create a *rewrite function*, which is presented with a flow-graph node and with the dataflow facts on the edges coming into that node (Section 6.2). The function either proposes to replace the node with a fresh subgraph, or it leaves the node alone. If the function proposes a replacement, the replacement must preserve semantics; preservation may be justified by incoming facts. For example, in Section 2.1 the fact `x == 7` justifies replacing `z = x + y` with `z = 7 + y`.

Table 1 shows how Hoopl interacts with your client code. Hoopl defines the types of control-flow graphs, lattice operations, transfer functions, and rewrite functions. All these types are parameterized by the types of nodes in the control-flow graph, which *you* get to define, so you can use Hoopl with many intermediate languages (Table 1). Function types are also parameterized by the type of dataflow facts, so you can define different analyses, using different types of facts, all operating over one type of graph.

To run an optimization, you pass lattice operations, transfer functions, and rewrite functions to one of Hoopl's *solver functions* or *rewrite functions*—Hoopl's *dataflow engine*. A solver function uses a forward or backward *analysis* to compute a dataflow fact for each program point (Section 3.3). A rewrite function uses a forward or backward *transformation* to compute facts and to rewrite a control-flow graph in light of those facts (Section 6).

```
data ChangeFlag = NoChange | SomeChange
data DataflowLattice a = DataflowLattice
 {fact_bot        :: a,
  fact_add_to     :: a -> a -> (a, ChangeFlag) }
```

Figure 2. Representation of a dataflow lattice

3.1 Dataflow lattices

As an example, we present a lattice of facts about constant propagation. At any program point, a standard constant-propagation analysis computes exactly one of three facts about a variable x:

- The analysis shows that $x = k$, where k is a compile-time constant of type `Const`.
- The analysis shows that x is *not* a compile-time constant. We notate this fact as $x = \top$.
- The analysis shows nothing about x, which we notate $x = \bot$.

The bottom element of the lattice is $x = \bot$, and the join operation $\sqcup$ approximates disjunction, the logical operation that combines facts flowing to a single label. A disjunction of two inconsistent facts is represented by $x = \top$, so for example $x = 7 \vee x = 8$ is approximated by $x = \top$, losing information.[2]

The lattice used by the analysis is the Cartesian product of the lattices for all the local variables. We represent this lattice as a finite map from a variable to a value of type `Maybe Const`. A variable x is not in the domain of the map iff $x = \bot$; x maps to `Nothing` iff $x = \top$; x maps to `Just` k iff $x = k$.

Hoopl's dataflow engine uses joins in a stylized way. Joins occur at labels. If f_{id} is the fact currently associated with the label id, and if a transfer function propagates a new fact f_{new} into the label id, the dataflow engine replaces f_{id} with the join $f_{new} \sqcup f_{id}$. Furthermore, the dataflow engine wants to know if $f_{new} \sqcup f_{id} = f_{id}$, because if not, the analysis has not reached a fixed point.

When computing a join, it is often cheap to learn if the join is equal to one of the arguments. We therefore use a nonstandard representation of lattice operations, as shown in Figure 2. The join operation $\sqcup$ and equality test $=$ are represented by a single function called `fact_add_to`. The term `fact_add_to` f_{new} f_{id} is equal to $(f_{id}, \texttt{NoChange})$ if $f_{new} \sqcup f_{id} = f_{id}$ and is equal to $(f_{new} \sqcup f_{id}, \texttt{SomeChange})$ otherwise. The `fact_bot` value is the bottom element.

3.2 Transfer functions

A transfer function is presented with dataflow facts on edges coming into a node, and it computes dataflow facts on outgoing edges. To understand transfer functions, we must understand how Hoopl organizes the nodes and edges of a control-flow graph.

A control-flow graph is a collection of *basic blocks*, each labelled with a `BlockId`. A basic block is a sequence beginning with a *first node*, containing zero or more *middle nodes*, and ending in a *last node*. (An optimizer also works with *subgraphs*, which, as discussed in Section 6.1, may omit an initial first node or a final last node.) A first node is always a `BlockId`; a typical middle node assigns to a register or memory location; and a typical last node is a conditional, unconditional, or indirect branch. You choose the types of middle and last nodes to suit your intermediate representation; if these types are `m` and `l`, the type of a basic block is `Block m l`.

[2] Your client code determines how much information is lost. For example, in a similar analysis for a functional language, you might track whether a value is the result of applying a constructor from any finite set $\{C_i\}$.

```
newtype LastOuts a = LastOuts [(BlockId, a)]
data ForwardTransfers m l a = ForwardTransfers
 {ft_first_out  :: BlockId -> a -> a,
  ft_middle_out :: m       -> a -> a,
  ft_last_outs  :: l       -> a -> LastOuts a}

data BackTransfers m l a = BackTransfers
 {bt_first_in  :: BlockId -> a               -> a,
  bt_middle_in :: m       -> a               -> a,
  bt_last_in   :: l       -> (BlockId -> a) -> a}
```

Figure 3. Transfer functions for forward and backward analyses.

First nodes are the only targets of control transfers; middle nodes never perform control transfers; and last nodes always perform control transfers. So a first node has arbitrarily many predecessors and exactly one successor; a middle node has exactly one predecessor and one successor; and a last node has exactly one predecessor and arbitrarily many successors.

These constraints on number of predecessors and successors determine the signatures of transfer functions, which are shown in Figure 3. For each type of node (first, middle, last) and for each kind of analysis (forward, backward), there is a distinct transfer function. Functions are grouped by kind of analysis, and each group is parameterized over a dataflow fact of type `a` and over the types `m` and `l` of middle and last nodes.

A fact in a forward analysis typically represents an assertion about program state, and because a label does not change program state, the transfer function `ft_first_out` is often `flip const`—a variation on the identity function.[3] For a middle node, the transfer function `ft_middle_out` is given a node and a precondition and returns an approximation of the strongest postcondition. For a last node, different postconditions may be propagated to different successors; for example, the true and false successors of a conditional branch may accumulate information implied by the truth or falsehood of the condition. A collection of (successor, fact) pairs is represented by a value of type `LastOuts a` (Figure 3).

In a forward analysis, the dataflow engine starts with the fact at the beginning of a block and applies transfer functions to the nodes in that block until eventually the transfer function for the last node computes the facts that are propagated to the block's successors. For example, in the block

```
L1: x = 7;
    y = 8;
    z = x + y;
    goto L2;
```

a forward analysis would propagate the fact $\mathtt{x} = 7 \wedge \mathtt{y} = 8$, which we will call f_{new}, along the edge to L2. The dataflow engine then *replaces* the current fact at L2 ($f_{\mathtt{L2}}$) with the lattice join $f_{new} \sqcup f_{\mathtt{L2}}$. The dataflow engine iterates over the blocks repeatedly, creating new facts f and joining them with facts f_{id} until $f \sqcup f_{id} = f_{id}$ at every label id. When the facts at labels stop changing, the dataflow engine has reached a fixed point.

3.3 Running the dataflow engine

Given lattice operations of type `DataflowLattice a` (Figure 2) together with transfer functions of type `ForwardTransfers m l a` (Figure 3), you can run the corresponding analysis by calling Hoopl function `zdfSolveFwd`, which is a part of our dataflow engine (a backward analysis calls function `zdfSolveBwd`, which has a similar type):

```
zdfSolveFwd
  :: HavingSuccessors l      -- Find successors of l
  => PassName                -- Name of the analysis
  -> DataflowLattice a       -- Lattice
  -> ForwardTransfers m l a -- Transfer functions
  -> a                       -- Input fact
  -> Graph m l               -- Control-flow graph
  -> FwdFixedPoint m l a ()
```

The function is polymorphic in the types of middle and last nodes `m` and `l` and in the type of the dataflow fact `a`. Polymorphism allows Hoopl to work with any intermediate language, as long as the type of last node `l` satisfies the constraint `HavingSuccessors l` by providing a function `succs` of type `l -> [BlockId]`, which gives the labels of the blocks to which a last node of type `l` might transfer control.

After the type constraint, the first three arguments to `zdfSolveFwd` characterize the analysis. The next argument is the dataflow fact that holds on entry to the graph; because a procedure's caller may establish some facts about parameters or about the stack, this fact is not always $\perp$. The last argument to `zdfSolveFwd` is the graph, and the result is a fixed point.

The `FwdFixedPoint` data structure, whose final type parameter `()` is explained in Section 6.3, is a big bag of information about a solution. The most significant information is a finite map from each block label to the dataflow fact that holds at the label, which is extracted using function `zdfFpFacts`:

```
type BlockEnv a = Data.Map BlockId a
zdfFpFacts :: FwdFixedPoint m l a g -> BlockEnv a
```

4. Related work

While dataflow analysis and optimization are covered by a vast literature, *design* of optimizers, the topic of this paper, is covered relatively sparsely. We therefore focus on foundations.

When transfer functions are monotone and lattices are finite in height, iterative dataflow analysis converges to a fixed point (Kam and Ullman 1976). If the lattice's join operation distributes over transfer functions, this fixed point is equivalent to a join-over-all-paths solution to the recursive dataflow equations (Kildall 1973).[4] Kam and Ullman (1977) generalize to some monotone functions. Each client of Hoopl must guarantee monotonicity, but for transfer functions that approximate weakest preconditions or strongest postconditions, monotonicity falls out naturally.

Cousot and Cousot (1977) introduce abstract interpretation as a technique for developing lattices for program analysis. Schmidt (1998) shows that an all-paths dataflow problem can be viewed as model checking an abstract interpretation.

The soundness of interleaving analysis and transformation, even when some speculative transformations are not performed on later iterations, was shown by Lerner, Grove, and Chambers (2002).

5. Example analysis passes

Hoopl makes it easy to write compiler passes based on dataflow. To show *how* easy, we present two analyses; related transformations appear in Section 6. The examples help solve a real problem in the Glasgow Haskell Compiler: because most calls are tail calls, GHC

[3] Not every fact is about program state, so not every forward analysis can ignore labels. For example, dominator analysis and other all-paths analyses often compute a set of labels through which control may (or must) pass.

[4] Kildall uses meets, not joins. Lattice orientation is conventional, and conventions have changed. We use Dana Scott's orientation, in which higher elements carry more information.

Dataflow fact and operations

```
data AvailVars = UniverseMinus VarSet | AvailVars VarSet
extendAvail  :: AvailVars -> LocalVar  -> AvailVars  -- add var to set
delFromAvail :: AvailVars -> LocalVar  -> AvailVars  -- remove var from set
elemAvail    :: AvailVars -> LocalVar  -> Bool       -- set membership
interAvail   :: AvailVars -> AvailVars -> AvailVars  -- set intersection
smallerAvail :: AvailVars -> AvailVars -> Bool       -- compare sizes
```

Lattice

```
availVarsLattice :: DataflowLattice AvailVars
availVarsLattice = DataflowLattice empty add
    where empty = UniverseMinus emptyVarSet
          add new old = let join = interAvail new old in
                        (if join `smallerAvail` old then SomeChange else NoChange, join)
```

Transfer functions

```
availTransfers :: ForwardTransfers CmmMiddle CmmLast AvailVars
availTransfers = ForwardTransfers (flip const) middleAvail lastAvail

middleAvail :: CmmMiddle -> AvailVars -> AvailVars
middleAvail (MidAssign (CmmLocal x) (CmmLoad l) avail | l `isStackSlotOf` x = extendAvail avail x
middleAvail (MidAssign lhs _expr) avail = foldVarsDefd delFromAvail avail lhs
middleAvail (MidStore l (CmmVar (CmmLocal x))) avail | l `isStackSlotOf` x = avail
middleAvail (MidStore l _) avail | isStackSlot l = delFromAvail avail (varOfSlot l)
middleAvail (MidStore _ _) avail = avail

lastAvail :: CmmLast -> AvailVars -> LastOuts AvailVars
lastAvail (LastCall _ (Just k) _ _) _ = LastOuts [(k, AvailVars emptyVarSet)]
lastAvail l avail = LastOuts $ map (\id -> (id, avail)) $ succs l
```

Available-variables analysis

```
cmmAvailableVars :: Graph CmmMiddle CmmLast -> BlockEnv AvailVars
cmmAvailableVars g = zdfFpFacts fp
  where fp = zdfSolveFwd "available variables" availVarsLattice
                  availTransfers (fact_bot availVarsLattice) g
```

Figure 4. Dataflow analysis pass to compute available variables

uses no callee-saves registers. Therefore, at each (rare) non-tail call, all live variables must be spilled to the stack.

To illustrate the results of the example analyses and transformations, here is a contrived example program in the style of Section 2:

```
f (bits32 a) {
  bits32 w, x, y, z;  // local variables
  x = a * a;
  w = a + a + a;
  y = g(w);           // call; x must be spilled
  z = y + y;
  if (y > 0) {
    return z;
  } else {
    return z + x;
  }
}
```

A spill and a reload should be inserted as follows:

```
f (bits32 a) {
  bits32 w, x, y, z;
  x = a * a;
  SPILL x;
  w = a + a + a;    // no register pressure from x
  y = g(w);
  z = y + y;        // no register pressure from x
  if (y > 0) {
    return z;       // x does not need reloading
  } else {
    RELOAD x;
    return z + x;
  }
}
```

Although the `SPILL` and `RELOAD` operations are introduced because of the call to `g(a)`, they are moved as far from the call as possible: `x` is spilled immediately after being assigned `a * a`, and `x` is reloaded not immediately after the call to `g`, but just before its use in the expression `z + x`. On the control-flow path to `return z`, `x` needn't be reloaded at all.

Spills and reloads are inserted by a sequence of dataflow passes:

1. A backward analysis computes liveness to identify the variables that should be spilled at call sites (Section 5.3 and Figure 5). An accompanying transformation (not shown) inserts reloads immediately after each call site and inserts spills not immediately before call sites, but rather immediately after the reaching definitions.
2. A forward analysis finds "available variables" which have been reloaded from the stack (Section 5.2 and Figure 4), and an accompanying transformation inserts redundant reloads before their uses (Section 6.4 and Figure 8). By keeping variables on the stack longer, this pass reduces register pressure.
3. A backward analysis (the same as in pass 1) computes liveness, and an accompanying transformation (Figure 9 in Section 6.5), dead-assignment elimination, removes redundant reloads.

Passes 2 and 3 cooperate to "sink" reloads away from the call site.

5.1 Choosing node types for GHC

To show that Hoopl works at scale, we present examples that have been implemented and tested in GHC. GHC's low-level intermediate code, called `Cmm`, is a subset of the portable assembly language C-- (Ramsey and Peyton Jones 2000). We specialize Hoopl to GHC by instantiating type parameters `m` and `l` with GHC's types `CmmMiddle` and `CmmLast`.

Dataflow fact

```
type Live = VarSet
```

Lattice

```
liveLattice :: DataflowLattice Live
liveLattice = DataflowLattice emptyVarSet add
  where add new old =
          let join = unionVarSets new old in
          (if sizeVarSet join > sizeVarSet old then SomeChange else NoChange, join)
```

Transfer functions

```
liveTransfers :: BackTransfers CmmMiddle CmmLast Live
liveTransfers = BackTransfers (flip const) middleLiveness lastLiveness

middleLiveness :: CmmMiddle -> Live -> Live
lastLiveness   :: CmmLast -> (BlockId -> Live) -> Live
middleLiveness m = addUsed m . remDefd m
lastLiveness   l = addUsed l . remDefd l . lastLiveOut l

addUsed :: UserOfLocalVars    a => a -> Live -> Live
remDefd :: DefinerOfLocalVars a => a -> Live -> Live
addUsed a live = foldVarsUsed extendVarSet  live a
remDefd a live = foldVarsDefd delFromVarSet live a

lastLiveOut :: CmmLast -> (BlockId -> Live) -> Live
lastLiveOut l env = last l
  where last (LastBranch id)        = env id
        last (LastCondBranch _ t f) = unionVarSets (env t) (env f)
        last (LastSwitch _ tbl)     = unionManyVarSets $ map env (catMaybes tbl)
        last (LastCall { })         = emptyVarSet
```

Liveness analysis

```
cmmLiveness :: Graph CmmMiddle CmmLast -> BlockEnv Live
cmmLiveness g = zdfFpFacts fp
   where fp = zdfSolveBwd "liveness" liveLattice liveTransfers emptyVarSet g
```

Figure 5. Dataflow analysis pass to compute liveness

A middle node stores the value of an expression:

```
data CmmMiddle
  = MidAssign CmmVar  CmmExpr -- store in variable
  | MidStore  CmmExpr CmmExpr -- store in memory
```

Type `CmmVar` represents a variable, which may be local (`CmmLocal LocalVar`) or global (`CmmGlobal GlobalVar`). Type `CmmExpr` represents a pure expression; among its constructors are `CmmLoad` (a value from memory) and `CmmVar` (the value of a variable).

A last node represents a control transfer; constructors include unconditional, conditional, and indirect branches, as well as a call:

```
data CmmLast
  = LastBranch     BlockId
  | LastCondBranch CmmExpr BlockId BlockId
  | LastSwitch     CmmExpr [Maybe BlockId]
  | LastCall ...     -- arguments omitted
```

5.2 Available variables: a forward analysis supporting pass 2

To understand the available-variables analysis, you must know that each variable x is related to a stack slot s_x, which is used to save the value of x. (GHC represents the relation using Haskell functions `isStackSlot`, `varOfSlot`, and `isStackSlotOf`.) If the variable and the stack slot hold the same value, that is if $x = s_x$, then it is *safe* to insert a reload.

To sink a reload of a variable x, we insert redundant reloads immediately before uses of x. It is *profitable* to insert a reload before a use of x only if, on every path to the use, the most recent definition of x is a reload from s_x. Safety and profitability are incomparable; the dataflow fact computed by our analysis is the set we call *available* variables, for which it is safe *and* profitable to insert a reload. Because the assertion of interest is an "all-paths" property, the lattice-join operation is set intersection, and the bottom element is the universal set containing all variables.

Instead of the usual mutable bit vectors, we use a purely functional representation of sets—one in which we can represent the set of all variables without enumerating them. A set is either `UniverseMinus` s, which stands for all variables except those in the set s, or `AvailVars` s, which stands for the variables in the set s (Figure 4, line 1). The bottom element is `UniverseMinus emptyVarSet`. To manipulate these sets, we provide the functions declared in lines 2–6 of Figure 4.

The most interesting part of the analysis is the `middleAvail` transfer function in Figure 4.

- Line 15 identifies an assignment that reloads local variable `x` from its stack slot. After such an assignment, $x = s_x$, and the last definition of x is a reload, so `x` is added to the set of available variables.
- On line 16, an assignment to a local variable means that the variable need not be equal to the value in its stack slot, so if `lhs` is a local variable, it is removed from the set of available variables. The conditional removal is done by applying `foldVarsDefd` to `delFromAvail`; `foldVarsDefd` is an overloaded function which, along with its dual, is used throughout the back end:

  ```
  foldVarsUsed :: UserOfLocalVars a
          => (b -> LocalVar -> b) -> b -> a -> b
  foldVarsDefd :: DefinerOfLocalVars a
          => (b -> LocalVar -> b) -> b -> a -> b
  ```

  On line 16, if `lhs` is a local variable, `foldVarsDefd` calls `delFromAvail`; if `lhs` is global, `foldVarsDefd` does nothing.
- There are three cases for `MidStore` nodes. Line 17 matches a node that spills a variable `x` to the stack. After such a node, $\mathtt{x} = s_\mathtt{x}$, but the node is not a reload instruction, so `x` is not added to the set of available variables. Line 18 matches a node that writes any *other* value to a stack slot, after which the variable associated with that slot is no longer available. Line 19 matches

a store to a location that is not a stack slot, which leaves the set of available variables unchanged.

The transfer function for a last node checks to see if the node is a function call (line 21); if so, the set of available variables at the call's continuation is empty. Other last nodes do not change values of variables or stack slots, so the set of available variables remains unchanged. A first node has no effect on program state, so its transfer function is `flip const` (line 13).

Given the lattice and the transfer functions, we can perform the analysis by calling the Hoopl function `zdfSolveFwd` (Figure 4, lines 25–26). Except for the implementations of the set operations on lines 2–6, Figure 4 shows the *entire* analysis.

5.3 Liveness: a backward analysis supporting passes 1 and 3

The assertion computed by a backward dataflow analysis applies to a *continuation* at a program point. The classic example is liveness analysis; the assertion of interest is that at a particular program point, the answer produced by the continuation does not depend on the value of a particular variable x. If so, x is said to be *dead* at that point. If the answer produced by the continuation *might* depend on the value of x, x is *live*.[5]

In a modern compiler, liveness analysis supports many program transformations, including dead-assignment elimination, which removes assignments to dead variables, and register allocation, which ensures that if two variables are live at the same time, they are not assigned to the same register.

The dataflow fact we use to represent liveness assertions is the set of live variables (Figure 5, line 1). The bottom element of the lattice is the empty set, and the join operation is set union (Figure 5, lines 2–6); a variable is deemed live after a node if it is live on *any* edge leaving that node.

The transfer functions for liveness rely on two auxiliary functions `addUsed` and `remDefd` (Figure 5, lines 13–16). A transfer function is given a set of variables live on the edges going out of the node. It removes from that set any variable defined by the node, then adds any variable used by the node (Figure 5, lines 11 and 12).

For a last node, function `lastLiveOut` consults the solution in progress (parameter `env` on line 18) to find out what variables are live at the *successors* of a last node. For an unconditional branch, we look up the live set at the label branched to (line 19); for a conditional branch, we look at both true and false edges (line 20), and for a switch, we consider every possible target of the branch (line 21). The remaining case (line 22) is a call, and since a call destroys the values of all local variables, no local variables are live at its continuation.

Given the lattice and the transfer functions, we perform liveness analysis by calling the dataflow-engine function `zdfSolveBwd` (Figure 5, line 25). Figure 5 shows the *entire* analysis.

6. Using dataflow facts to rewrite graphs

We compute dataflow facts in order to enable code-improving transformations on control-flow graphs. A dataflow fact may enable a rewrite function to replace a node by a *subgraph*. A subgraph is a graph that may not define all the labels to which it refers. A valuable, novel property of our implementation is that it uses Haskell's static type system to control which subgraphs may replace which nodes. Before explaining how to transform graphs, we explain how graphs and subgraphs are represented.

[5] Liveness cannot be decided accurately; it reduces to the halting problem. As usual, we approximate liveness by reachability.

```
type O  -- marks graph as open   at entry or exit
type C  -- marks graph as closed at entry or exit
type GF m l entry exit -- graph or subgraph
type Graph m l = GF m l O C
```

Figure 6. Types of graphs and subgraphs

6.1 Representing graphs and subgraphs

As mentioned in Section 3.2, a graph is a collection of basic blocks, and a basic block is normally a first node followed by zero or more middle nodes followed by a last node. But a graph may also contain two special, incomplete blocks:

- A graph may begin with an *entry sequence*: zero or more middle nodes followed by a last node (i.e., a control transfer). Such a graph is *open at the entry*.
- A graph may end with an *exit sequence*: a first node followed by zero or more middle nodes, but *not* followed by a last node. Such a graph is *open at the exit* (control "falls off the end").

Our general type of graph, called `GF`, therefore takes *four* type parameters (Figure 6): `m` is the type of a middle node; `l` is the type of a last node; `entry` is either type `O` or type `C`, depending on whether the graph is open or closed at the entry; and `exit` is type `O` or type `C`, depending on whether the graph is open or closed at the exit. The instantiations of type parameters `entry` and `exit` specify the graph's *shape*, which we refer to in shorthand. For example, a full `Graph`, which represents a function or procedure, is open at the entry and closed at the exit, or simply "open/closed."

Graphs are created using these functions:

```
mkLabel    :: BlockId         -> GF m l C O
mkMiddle   :: m               -> GF m l O O
mkLast     :: l               -> GF m l O C
(<*>)      :: GF m l e a      -> GF m l a x
                              -> GF m l e x
emptyGraph :: GraphClosure a => GF m l a a
```

The infix `<*>` function is graph concatenation; the exit of the first argument must match the entry of the next (both open or both closed). The `emptyGraph` is a left and right unit of concatenation; the constraint `GraphClosure a` is satisfied only by types `O` and `C`.

A graph is normally represented by a triple: an optional entry sequence, a `BlockEnv` containing basic blocks, and an optional exit sequence. As a special case, a single sequence of middle nodes also forms a graph open at both entry and exit.

This new representation improves significantly on our previous work (Ramsey and Dias 2005):

- We can find the exit point of a graph in constant time.
- We can concatenate data structures in near-constant amortized time. Previously, we had to resort to Hughes's (1986) technique, representing a graph as a function.
- Most important, errors in concatenation are ruled out at compile-compile time by Haskell's static type system. In earlier implementations, such errors were not detected until the compiler ran, at which point Hoopl tried to compensate for the errors—but the compensation code harbored subtle faults.

6.2 Rewrite functions

Hoopl transforms its graphs by composing transfer functions (Section 3.2) with *rewrite functions*, whose types are shown in Figure 7. A rewrite function is given a dataflow fact and a node n. It may choose to replace node n with a *replacement graph* g, in which

```
type Rewrite m l e x = Maybe (GF m l e x)
data ForwardRewrites m l a = ForwardRewrites
 {fr_first  :: BlockId -> a -> Rewrite m l C O,
  fr_middle :: m       -> a -> Rewrite m l O O,
  fr_last   :: l       -> a -> Rewrite m l O C}

data BackwardRewrites m l a = BackwardRewrites
 {br_first  :: BlockId -> a      -> Rewrite m l C O,
  br_middle :: m       -> a      -> Rewrite m l O O,
  br_last   :: l -> (BlockId->a) -> Rewrite m l O C}
```

Figure 7. Types of forward and backward rewrite functions.

case it returns `Just` g, or it may do nothing, in which case it returns `Nothing`. If it returns `Just` g, it must guarantee that given the assertions represented by incoming dataflow facts, graph g is observationally equivalent to node n.

A rewrite function may replace a node only with a graph of the same shape:

- A first node must be rewritten to a closed/open graph.
- A middle node must be rewritten to an open/open graph.
- A last node must be rewritten to an open/closed graph.

These conditions, which are enforced by the static type system (Figure 7), are necessary and sufficient to ensure that every replacement graph can be spliced in place of the node it replaces.

6.3 Running the dataflow engine

To write a program transformation, you must

- Create a dataflow lattice and transfer functions for the supporting analysis, as described in Section 3.
- Create rewrite functions for first, middle, and last nodes.

You can then use Hoopl function `zdfRewriteFwd` to transform a control-flow graph (a backward transformation uses function `zdfRewriteBwd`, which has a similar type):

```
zdfRewriteFwd
 :: HavingSuccessors l     -- Find successors of l
 => RewritingDepth         -- Rewrite recursively?
 -> PassName               -- Name of this pass
 -> DataflowLattice a      -- Lattice
 -> ForwardTransfers m l a -- Transfer functions
 -> ForwardRewrites  m l a -- Rewrite functions
 -> a                      -- Input fact
 -> Graph m l              -- Graph or subgraph
 -> FuelMonad (FwdFixedPoint m l a (Graph m l))
```

Function `zdfRewriteFwd` is like `zdfSolveFwd` in Section 3.3, but it uses and produces extra information:

- Function `zdfRewriteFwd` requires rewrite functions as well as transfer functions.
- The `RewritingDepth` parameter controls recursive rewriting; if a graph produced by a rewrite function should not be further rewritten, rewriting is *shallow*; if a graph produced by a rewrite function can be rewritten again, rewriting is *deep*.
- In the result type, the fourth type parameter of type constructor `FwdFixedPoint` is a value contained in the fixed point. The value is extracted using function `zdfFpContents`, which has type `FwdFixedPoint m l a b -> b`. Here the type parameter `b` is instantiated to `Graph m l`: the fixed point contains the rewritten graph.
- Rewriting is monadic. A `FuelMonad` holds resources needed to rewrite nodes into subgraphs: a supply of fresh labels and a supply of *optimization fuel* (Section 7.1).

Function `zdfRewriteFwd` implements interleaved analysis and transformation in two phases (Lerner, Grove, and Chambers 2002):

- In the first phase, when a rewrite function proposes to replace a node n, the replacement graph is analyzed recursively, and the results of that analysis are used as the new dataflow fact(s) flowing out of node n. Then the replacement graph is *thrown away*; only the facts remain. (In other words, rewriting is *speculative*.) If, on a later iteration, node n is analyzed again, perhaps with a different input fact, the rewrite function may propose a different replacement or even no replacement at all.

  The first phase is called the *iterator*. It computes a fixed point of the dataflow analysis *as if* nodes were replaced, while never actually replacing a node.
- When the iterator finishes, the resulting fixed point is sound, and the facts in the fixed point are used by the second phase, in which no dataflow facts change, but rewrites are not speculative: each replacement proposed by a rewrite function is actually performed. This phase is therefore called the *actualizer*.

Facts computed by the iterator depend on graphs produced by rewrite functions, which in turn depend on facts computed by the iterator. How do we know this algorithm is sound, or even if it terminates? A proof requires its own POPL paper (Lerner, Grove, and Chambers 2002), but we can give some intuition:

- The algorithm is sound because, given the incoming dataflow facts, each rewrite must preserve the observable behavior of the program. A sound analysis of the rewritten graph may generate only dataflow facts that could have been generated by a more complicated analysis of the original graph.
- No matter what the transfer functions and rewrite functions do, the dataflow engine uses the dataflow lattice's join operation to ensure that facts at labels never decrease. As long as no fact may increase infinitely many times, analysis terminates.

Thus to guarantee soundness and termination, client code must supply sound transfer functions, sound rewrite functions, and a lattice with no infinite ascending chains. And unless client code specifies shallow rewriting, rewrite functions must not return replacement graphs which contain nodes that could be rewritten indefinitely.

Why use such a complex algorithm? Because interleaving analysis with transformation makes it possible to implement useful transformations using startlingly simple client code. In the rest of this section we present two examples: Section 6.4 shows how to insert a reload instruction just before each use of each spilled variable, and Section 6.5 shows how to eliminate dead assignments. When these two transformations are run in sequence, the effect is to sink reloads and produce programs like the example shown in Section 5.

6.4 Sinking reloads: a forward transformation

We use the available-variables analysis of Section 5.2 to insert reloads immediately before uses of variables. The transformation is implemented by the rewrite functions on lines 3–5 of Figure 8. A first node uses no variables and so is never rewritten. For middle and last nodes, `maybe_reload_before` (lines 6–9) computes `used`, which is the set of variables used in the node that are both safe and profitable to reload. If that set is not empty, function `reloadTail` replaces `node` with a new graph in which `node` is preceded by a (redundant) reload for each variable in the set `used`. A reload node is created by function `reload` (line 11), which has type `LocalVar -> CmmMiddle`.

```
1: availRewrites :: ForwardRewrites CmmMiddle CmmLast AvailVars
2: availRewrites = ForwardRewrites first middle last
3:   where first _ _ = Nothing
4:         middle m avail = maybe_reload_before avail m (mkMiddle m)
5:         last   l avail = maybe_reload_before avail l (mkLast l)
6:         maybe_reload_before avail node tail =
7:             let used = filterVarsUsed (elemAvail avail) node
8:             in  if isEmptyVarSet used then Nothing
9:                 else Just $ reloadTail used tail
10:        reloadTail vars t = foldl rel t $ varSetToList vars
11:            where rel t r = mkMiddle (reload r) <*> t
```

Rewrite functions

```
12: insertLateReloads :: Graph CmmMiddle CmmLast -> FuelMonad (Graph CmmMiddle CmmLast)
13: insertLateReloads g = liftM zdfFpContents fp
14:   where fp = zdfRewriteFwd RewriteShallow "insert late reloads" availVarsLattice
15:                  availTransfers availRewrites (fact_bot availVarsLattice) g
```

Late-reload insertion

Figure 8. Late-reload insertion, which relies on the analysis of Figure 4

```
1: deadRewrites = BackwardRewrites nothing middleRemoveDeads nothing
2:   where nothing _ _ = Nothing
3:         middleRemoveDeads :: CmmMiddle -> VarSet -> Maybe (Graph CmmMiddle CmmLast)
4:         middleRemoveDeads (MidAssign (CmmLocal x) _) live
5:             | not (x `elemVarSet` live) = Just emptyGraph
6:         middleRemoveDeads _ _ = Nothing
```

Rewrite functions

```
7: removeDeadAssignments :: Graph CmmMiddle CmmLast -> FuelMonad (Graph CmmMiddle CmmLast)
8: removeDeadAssignments g = liftM zdfFpContents fp
9:      where fp = zdfRewriteBwd RewriteDeep "dead-assignment elim" liveLattice
10:                     liveTransfers deadRewrites emptyVarSet g
```

Dead-code elimination

Figure 9. Dead-assignment elimination, which relies on the analysis of Figure 5

Our transformation is implemented by the call to `zdfRewriteFwd` on lines 14–15 of Figure 8. Rewriting is shallow, so a graph containing reload nodes is not itself rewritten. (If it *were* rewritten, a nonempty `used` set would make the compiler insert an infinite sequence of reloads before `node`.) Once the reloads are inserted, the original reloads are dead, and they can be eliminated by our next transformation, dead-assignment elimination.

6.5 Dead-assignment elimination: a backward transformation

We use the liveness analysis of Section 5.3 to identify assignments to local variables that are not live. Such *dead assignments* can be removed without changing the observable behavior of the program. The removal is implemented by the rewrite functions on lines 2–6 of Figure 9. First and last nodes are not assignments and so are never rewritten. A middle node is rewritten to the empty graph if and only if it is an assignment to a dead variable (lines 4–5). On lines 9 and 10, we call `zdfRewriteBwd`. That's the whole thing.

7. Hoopl's dataflow engine

In sections 3 through 6, we use Hoopl to create analyses and transformations. Here we sketch the implementation of the main part of Hoopl: the dataflow engine. While a full description of the implementation is beyond the scope of this paper, a sketch demonstrates the new ideas that make this implementation simpler than the original: using pure functional code throughout; using an explicit state monad to manage the computation of fixed points; giving each type of graph node its own analysis function, which also performs speculative rewriting; and using continuation-passing style to stitch these functions together. We sketch the implementation from the bottom up: Hoopl's fuel monad, the monad that holds dataflow facts, an iterator, and an actualizer.

7.1 Throttling the dataflow engine using "optimization fuel"

We have extended Lerner, Grove, and Chambers's optimization-combining algorithm with Whalley's (1994) algorithm for isolating faults. Whalley's algorithm is used to test a faulty optimizer; it automatically finds the first rewrite that introduces a fault in a test program. It works by giving the optimizer a finite supply of *optimization fuel*. Each time a rewrite function proposes to replace a node, one unit of fuel is consumed. When the optimizer runs out of fuel, further rewrites are suppressed. Because each rewrite leaves the observable behavior of the program unchanged, it is safe to suppress rewrites at any point. In normal operation, the optimizer has unlimited fuel, but during debugging, a fault can be isolated quickly by doing a binary search on the size of the fuel supply. The fuel supply is stored in a state monad (`FuelMonad`), which also holds a supply of fresh labels. Fresh labels are used for making new blocks.

7.2 A monad for dataflow effects

In addition to fuel, each analysis and transformation keeps track of the values of dataflow facts. Facts and fuel are stored in a *dataflow monad*, a state-transformer monad whose state includes a private *environment* mapping labels to facts, as well as the global supplies of fuel and fresh labels. A value in the dataflow monad has type `DFM a b`, where `a` is the type of a dataflow fact and `b` is the type of the value returned by the monadic action.

Operations on the dataflow monad include

```
getFact          :: BlockId ->      DFM a a
setFact          :: BlockId -> a -> DFM a ()
getAllFacts      :: DFM a (BlockEnv a)
setAllFacts      :: BlockEnv a -> DFM a ()
useOneFuel       :: DFM a ()
```

```
1: type FactKont a b = a          -> DFM a b
2: type LOFsKont a b = LastOuts a -> DFM a b
3: type Kont     a b =               DFM a b
4: fwd_iter :: forall m l e x a . HavingSuccessors l => (forall b . Maybe b -> DFM a (Maybe b))
5:         -> RewritingDepth -> PassName -> BlockEnv a -> ForwardTransfers m l a
6:         -> ForwardRewrites m l a -> ZMaybe e a -> GF m l e x -> DFM a (ZMaybe x a)
7: fwd_iter with_fuel depth name start_facts transfers rewrites in_fact g =
8:      do { setAllFacts start_facts ; iter_ex g in_fact }
9:    where iter_ex    :: GF m l e x -> ZMaybe e a -> DFM a (ZMaybe x a)
10:         iter_first :: BlockId            -> FactKont a b -> Kont     a b
11:         iter_mid   :: m                  -> FactKont a b -> FactKont a b
12:         iter_last  :: l                  -> LOFsKont a b -> FactKont a b
13:         iter_block :: BlockId -> [m] -> l -> LOFsKont a b -> Kont     a b
14:         iter_block f ms l = iter_first f . flip (foldr iter_mid) ms . iter_last l
15:         set_last :: LOFsKont a ()
16:         set_last (LastOuts l) = mapM_ (uncurry setFact) l
17:         iter_mid m k in' =
18:           (with_fuel $ fr_middle rewrites in' m) >>= \x -> case x of
19:             Nothing -> k (ft_middle_out transfers in' m)
20:             Just g  -> do { a <- subAnalysis $ case depth of
21:                                                  RewriteDeep    -> iter_OO g return in'
22:                                                  RewriteShallow -> anal_f_OO g in'
23:                           ; k a }
```

Figure 10. Excerpts from the forward iterator

```
fuelExhausted     :: DFM a Bool
subAnalysis       :: DFM a b -> DFM a b
withDuplicateFuel :: DFM a b -> DFM a b
runDFM :: DataflowLattice a -> DFM a b -> FuelMonad b
```

A computation in the dataflow monad has two significant side effects: it may *increase stored facts* (according to a lattice ordering) and it may *consume fuel*. The two most interesting operations in the monad are used to control those effects:

- Computation `subAnalysis c` computes the same results as `c` and consumes the same fuel as `c`, but it does not change any stored dataflow facts.
- Computation `withDuplicateFuel c` computes the same results as `c` and changes the same stored facts as `c`, but it consumes fuel from a *copy* of the fuel supply. The inner computation `c` may run out of fuel, but afterward, `withDuplicateFuel` restores the original fuel supply. Using `withDuplicateFuel` has enabled us to eliminate fuel from arguments and results, making an implementation which is less error-prone and *much* easier to read than the one by Ramsey and Dias (2005).

Function `runDFM` runs a single analysis or transformation, then abandons the dataflow facts and returns the result in the fuel monad. Only `FuelMonad` is exposed to the client; the dataflow monad is private to Hoopl. Using the dataflow monad, Hoopl's iterators and actualizers are significantly simpler than those in our previous work (Ramsey and Dias 2005). In Sections 7.3 and 7.4, we show parts of the forward iterator and actualizer.

7.3 The forward iterator

An *iterator* does dataflow analysis with speculative rewriting. Analysis begins an dataflow monad whose environment maps all labels to bottom facts. For each block in the control-flow graph, the iterator begins with the dataflow facts flowing into one end of the block (in a forward analysis, the first node; in a backward analysis, the last node), then uses the transfer functions and rewrite functions to compute the dataflow facts flowing out the other end of the block. The outflowing facts are joined with the facts previously stored in the environment, and when the facts in the environment stop changing, the iterator terminates.

The iterator interleaves analysis and speculative rewriting (Lerner, Grove, and Chambers 2002). At a node n, the iterator passes n and any incoming dataflow facts *fs* to a rewriting function. If node n is rewritten to a graph g, the iterator continues with the same dataflow facts *fs* flowing into graph g. After graph g is analyzed, it is discarded; only the facts flowing out of g persist.

Figure 10 shows excerpts from the forward iterator `fwd_iter`.

- The `with_fuel` parameter is called on the result of each rewriting function (e.g. line 18). It consumes fuel; or if no fuel is available, it prevents any nodes from being rewritten.
- Analysis of a subgraph starts with known facts, not bottom facts; they are passed as `start_facts` and set on line 8.
- A forward analysis requires an entry fact `in_fact` if and only if the graph being analyzed is open at the entry. Similarly, the analysis produces an output fact if and only if the graph being analyzed is open at the exit. We express these constraints using the generalized algebraic data type `ZMaybe` (Figure 10, line 6):

  ```
  data ZMaybe ex a where
    ZJust    :: a -> ZMaybe O a
    ZNothing ::      ZMaybe C a
  ```

  Using `ZMaybe` to construct the types of the input and output facts has simplified our implementation of the dataflow engine and has eliminated dynamic tests of the shapes of subgraphs.

The function `iter_ex` (type on line 9, implementation not shown), solves a graph or subgraph `g`. Where the graph is open, `iter_ex` converts `ZMaybe` facts to actual facts—the static type system precludes the possibility of a missing or superfluous fact.

The iterator is composed of functions written in continuation-passing style: the result of analyzing part of a graph is a function from continuations to continuations. The types of the continuations are shown on lines 1–3 of Figure 10.

```
1: type GraphFactKont  m l e x a b = GF m l e x -> a -> DFM a b
2: type GraphKont      m l e x a b = GF m l e x      -> DFM a b
3:       ar_first :: BlockId -> GraphFactKont m l e O a b -> GraphKont     m l e C a b
4:       ar_mid   :: m       -> GraphFactKont m l e O a b -> GraphFactKont m l e O a b
5:       ar_last  :: l       -> GraphKont     m l e C a b -> GraphFactKont m l e O a b
6:       ar_mid m k head in' =
7:         (with_fuel $ fr_middle rewrites in' m) >>= \x -> case x of
8:           Nothing -> k (head <*> mkMiddle m) (ft_middle_out transfers in' m)
9:           Just g  -> do { (g, a) <- subAnalysis $
10:                                case depth of
11:                                  RewriteDeep    -> iar_OO g (curry return) in'
12:                                  RewriteShallow -> do { a <- anal_f_OO g in'; return (g, a) }
13:                           ; k (head <*> g) a }
14:       iar_OO :: GF m l O O -> GraphFactKont m l O O a b -> FactKont a b
```

Figure 11. Excerpts from the forward actualizer

- Type `FactKont a b` describes a context following a first node or middle node: in a forward analysis, the context expects a fact of type `a` to flow out of the node. The rest of the analysis consumes that fact and produces a computation in the dataflow monad (`DFM a`) with an answer of type `b`.
- Type `LOFsKont a b` describes a context following a last node. The type is dictated by the type of the transfer function `ft_last_outs` in Figure 3: since as many facts flow out of a last node as there are control-flow edges leaving that node, the context expects those facts to have type `LastOuts a`.
- Type `Kont a b` describes a context *before* a first node (or a basic block). The dataflow fact flowing into the note is not passed as a parameter; it is extracted from the dataflow monad's environment by calling the monadic operation `getFact`.

Declarations of continuation-passing iterator functions for nodes are shown on lines 10–12 of Figure 10. Function `iter_last` on line 12 maps a `LOFsKont` to a `FactKont`; `iter_mid` on line 11 maps a `FactKont` to another `FactKont`; and `iter_first` on line 10 maps a `FactKont` to a `Kont`. To analyze a basic block, with speculative rewriting, we compose these three functions, as shown in function `iter_block` on lines 13 and 14.[6]

In code not shown here, function `iter_block` is applied to continuation `set_last` (lines 15 and 16), which updates the environment of facts stored in the dataflow monad. The value `iter_block set_last` is a computation of type `Kont a ()`, which is `DFM a ()`. This computation reads the stored fact flowing into a block, propagates facts through the block using transfer functions and speculative rewriting, and finally updates the stored facts flowing out to the block's successors. Iterator functions for graphs and subgraphs, like `iter_ex`, perform such a computation for every block, then repeat until stored facts stop changing. Each iteration runs under `withDuplicateFuel`, so `fwd_iter` *simulates* the effects of a fuel limit, but it does not actually consume fuel.

Computations in `fwd_iter`, such as `iter_block`, are compositions of `iter_first`, `iter_mid`, and `iter_last`. Because these three functions so resemble one another, we show only one: `iter_mid`, on lines 17–23 of Figure 10. On line 18, a rewrite function gets an input fact `in'` and a middle node `m`. If the rewrite function proposes no replacement graph, or if no fuel is available, the application of `with_fuel` returns `return Nothing`, and continuation `k` is given the output fact (computed by `ft_middle_out` on line 19). The interesting case occurs on lines 20–23, when the rewrite function proposes a replacement graph `g`. Function `with_fuel` decrements the fuel supply and produces `g`.

[6] To simplify the example, we conceal Hoopl's representation of blocks.

1. If we are doing *deep* rewriting, then as `g` is analyzed, it may be rewritten further. Because `g` replaces a middle node, it is open at entry and exit, so it is analyzed and rewritten on line 21 by a recursive call to `iter_OO` (implementation not shown; type `GF m l O O -> FactKont a b -> FactKont a b`). The recursive call gets continuation `return`, and the resulting `FactKont a a` is given the input fact. The output fact is computed in a sub-analysis and bound on line 20. Function `subAnalysis` rolls back the facts mutated by `iter_OO`, but `subAnalysis` does account for fuel consumed by `iter_OO`.
2. If we are doing *shallow* rewriting, the new graph `g` must not be rewritten, but we must still find a fixed point of the transfer equations. We compute that fixed point using `anal_f_OO` (line 22). Function `anal_f_OO` (not shown) recursively calls `fwd_iter` using `with_fuel = \ _-> return Nothing`, and so it does no rewriting and consumes no fuel.

Whether rewriting is shallow or deep, the application on line 23 solves the rest of the graph by applying the continuation `k`.

Function `zdfSolveFwd` is implemented by calling `fwd_iter` with the transfer functions given, with undefined rewrite functions, and with parameter `with_fuel = \ _ -> return Nothing`.

7.4 The forward actualizer

An iterator returns dataflow facts, leaving the graph unchanged. An *actualizer* takes facts and a graph, and in a single pass uses rewrite functions to create a new graph. The actualizer also uses transfer functions to materialize facts on edges within basic blocks.

The forward actualizer closely resembles the forward iterator, but because the actualizer passes a rewritten graph as an accumulating parameter, the continuations have different types, as shown on lines 1 and 2 of Figure 11. When the actualizer runs, the dataflow monad already contains a fixed point, so there is no need to propagate facts out of a block, and so no continuation analogous to `LOFsKont`.

The functions that actualize rewrites are again in continuation-passing style; lines 3–5 of Figure 11 give the types of the base-case functions. We show only `ar_mid` (lines 6–13). It is much like function `iter_mid` on lines 17–23 of Figure 10. Line 6 shows the additional parameter `head`, which contains the (rewritten) graph preceding middle node `m`. When no rewrite is proposed, the only change to the code is that continuation `k` takes the additional parameter `head <*> mkMiddle m`, which is the graph formed by concatenating graph `head` and node `m`. When a rewrite is proposed, the sub-analysis computes not just an output fact but also a possibly rewritten graph (lines 9–12). Rewriting proceeds with the new graph `head <*> g` (line 13).

The recursive iterate-and-actualize-rewrites function `iar_OO` (type on line 14, implementation not shown) has no counterpart in the iterator. It calls the iterator to set the dataflow facts to a fixed point (using a duplicate fuel supply), then calls actualize-rewrite functions to rewrite the graph based on those facts (using the shared fuel supply). Similar functions apply to graphs of other shapes; for example, `iar_OC` is used to implement `zdfRewriteFwd`.

8. Conclusions

Compiler textbooks make dataflow optimization appear difficult and complicated. In this paper, we show how to engineer a library, Hoopl, which makes it easy to build analyses and transformations based on dataflow. Hoopl makes dataflow simple not by using a single magic ingredient, but by applying ideas that are well understood by the programming-language community.

- We acknowledge only one program-analysis technique: the solution of recursion equations over assertions. We solve the equations by iterating to a fixed point.
- We consider only two ways of relating assertions: weakest liberal precondition and strongest postcondition, which correspond to "backward" and "forward" dataflow problems.
- Although our implementation allows graph nodes to be rewritten in any way that preserves semantics, we describe three program-transformation techniques: substitution of equals for equals, insertion of assignments to unobserved variables, and removal of assignments to unobserved variables (Section 2). Substitution of equals for equals is often justified by properties of program states; for example, if variable x is always 7, we may substitute 7 for x. Insertion and removal of assignments are often justified by properties of paths through programs; for example, if an assignment's continuation does not use the variable assigned to, that assignment may be removed.
- Complex program transformations should be composed from simple transformations. For example, both "code motion" and "induction-variable elimination" can be implemented in three stages: insert new assignments; substitute equals for equals; remove unneeded assignments (Section 2.2).
- Because each rewrite leaves the semantics of the program unchanged, we can use "optimization fuel" to limit the number of rewrites. When we isolate a fault (Section 7.1), we have to debug just a single rewrite, not a complex transformation.

We also build on proven implementation techniques in a way that makes it easy to implement classic code improvements.

- We use the algorithm of Lerner, Grove, and Chambers (2002) to compose analyses and transformations. This algorithm makes it easy to compose complex transformations from simple ones.

  Using continuation-passing style and generalized algebraic data types, we have created a new implementation, which works by composing three relatively simple functions (Section 7.3). The functions are simple because the static type of a node constrains the number of predecessors and successors it may have. And because we can compare our code with a standard continuation semantics, we have more confidence in this new implementation than in any previous implementation.
- Our code is pure. Inspired by Huet's (1997) zipper, we use an applicative representation of control-flow graphs (Ramsey and Dias 2005). We improve on our prior work by storing changing dataflow facts in an explicit dataflow monad, which makes it especially easy to implement such operations as sub-analysis of a replacement graph (Section 7.2); by using static types to guarantee that each replacement graph can be spliced in place of the node it replaces (Sections 6.1 and 6.2); and by simplifying our implementation using continuation-passing style (Sections 7.3 and 7.4).
- Hoopl is polymorphic in the representations of assignments and control-flow operations. By forcing us to separate concerns, introducing polymorphism made the code simpler, easier to understand, and easier to maintain. In particular, it forced us to make explicit *exactly* what Hoopl must know about flow-graph nodes: it must be able to find targets of control-flow operations (constraint `HavingSuccessors l`, Section 3.3).

Using Hoopl, you can create a new code improvement in three steps: create a lattice representation for the assertions you want to express; create transfer functions that approximate weakest preconditions or strongest postconditions; and create rewrite functions that use your assertions to justify program transformations. You can get quickly to the real intellectual work of code improvement: identifying interesting transformations and the assertions that justify them.

References

Manuel E. Benitez and Jack W. Davidson. 1988 (July). A portable global optimizer and linker. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 23(7):329–338.

Cliff Click and Keith D. Cooper. 1995 (March). Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems*, 17(2):181–196.

Patrick Cousot and Radhia Cousot. 1977 (January). Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252.

Gérard Huet. 1997 (September). The Zipper. *Journal of Functional Programming*, 7(5):549–554.

R. John Muir Hughes. 1986 (March). A novel representation of lists and its application to the function "reverse". *Information Processing Letters*, 22(3):141–144.

John B. Kam and Jeffrey D. Ullman. 1976. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171.

John B. Kam and Jeffrey D. Ullman. 1977. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317.

Gary A. Kildall. 1973 (October). A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 194–206.

Sorin Lerner, David Grove, and Craig Chambers. 2002 (January). Composing dataflow analyses and transformations. *Conference Record of the 29th Annual ACM Symposium on Principles of Programming Languages,* in *SIGPLAN Notices*, 31(1):270–282.

Norman Ramsey and João Dias. 2005 (September). An applicative control-flow graph based on Huet's zipper. In *ACM SIGPLAN Workshop on ML*, pages 101–122.

Norman Ramsey and Simon L. Peyton Jones. 2000 (May). A single intermediate language that supports multiple implementations of exceptions. *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 35 (5):285–298.

David A. Schmidt. 1998. Data flow analysis is model checking of abstract interpretations. In ACM, editor, *Conference Record of the 25th Annual ACM Symposium on Principles of Programming Languages*, pages 38–48.

David B. Whalley. 1994 (September). Automatic isolation of compiler errors. *ACM Transactions on Programming Languages and Systems*, 16 (5):1648–1659.

A. Index of defined identifiers

This appendix lists every nontrivial identifier used in the body of the paper. For each identifier, we list the page on which that identifier is defined or discussed—or when appropriate, the figure (with line number where possible). For those few identifiers not defined or discussed in text, we give the type signature and the page on which the identifier is first referred to.

Some identifiers used in the text are defined in the Haskell Prelude; for those readers less familiar with Haskell, these identifiers are listed in Appendix B.

B. Identifiers defined in Haskell Prelude

!, $, &, &&, *, +, ++, -, ., /, ==, >, >=, >>, >>=, Bool, const, curry, Data.Map, False, flip, foldl, foldr, fst, head, id, Int, Just, liftM, map, mapM_, Maybe, not, Nothing, return, snd, String, tail, True, uncurry, undefined .