Hoopl: A Modular, Reusable Library for Dataflow Analysis and Transformation

Programming pearl

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

Dataflow analysis and transformation of control-flow graphs is pervasive in optimizing compilers, but it is typically tightly interwoven with the details of a *particular* compiler. We describe Hoopl, a reusable Haskell library that makes it unusually easy to define new analyses and transformations for *any* compiler. Hoopl's interface is modular and polymorphic, and it offers unusually strong static guarantees. The implementation is also far from routine: it encapsulates state-of-the-art algorithms (interleaved analysis and rewriting, dynamic error isolation), and it cleanly separates their tricky elements so that they can be understood independently.

1. Introduction

A mature optimizing compiler for an imperative language includes many analyses, the results of which justify the optimizer's code-improving transformations. Many of the most important analyses and transformations—constant propagation, live-variable analysis, inlining, sinking of loads, and so on—should be regarded as particular cases of a single general problem: *dataflow analysis and optimization*. Dataflow analysis is over thirty years old, but a recent, seminal paper by Lerner, Grove, and Chambers (2002) goes further, describing a powerful but subtle way to *interleave* analysis and transformation so that each piggybacks on the other.

Because optimizations based on dataflow analysis share a common intellectual framework, and because that framework is subtle, it it tempting to try to build a single reusable library that embodies the subtle ideas, while making it easy for clients to instantiate the library for different situations. Tempting, but difficult. Although some such frameworks exist, as we discuss in Section 6, they have complex APIs and implementations, and none implements the Lerner/Grove/Chambers technique.

In this paper we present Hoopl (short for "higher-order optimization library"), a new Haskell library for dataflow analysis and optimization. It has the following distinctive characteristics:

- Hoopl is purely functional. Perhaps surprisingly, code that manipulates control-flow graphs is easier to write, and far easier to write correctly, when written in a purely functional style (Ramsey and Dias 2005). When analysis and rewriting are interleaved, so that rewriting must be done *speculatively,* without knowing whether the result of the rewrite will be retained or discarded, the benefit of a purely functional style is intensified (Sections 2 and 4.8).
- Hoopl is polymorphic. Just as a list library is polymorphic in the list elements, so is Hoopl polymorphic, both in the nodes that inhabit graphs, and in the dataflow facts that analyses compute over these graphs (Section 4).
- The paper by Lerner, Grove, and Chambers is inspiring but abstract. We articulate their ideas in a concrete but simple API that hides a subtle implementation (Sections 3 and 4). You provide a representation for assertions, a transfer function that transforms assertions across a node, and a rewrite function that uses a assertion to justify rewriting a node. Hoopl "lifts" these node-level functions to work over control-flow graphs, sets up and solves recursion equations, and interleaves rewriting with analysis. Designing good abstractions (data types, APIs) is surprisingly hard; we have been through over a dozen significantly different iterations, and we offer our API as a contribution.
- Analyses and transformations built on Hoopl are small, simple, and easy to get right because the client only has to perform local reasoning ("`y` is live before `x:=y+2`").[1] Moreover, Hoopl helps you write correct optimizations: it statically rules out transformations that violate invariants of the control-flow graph (Sections 3 and 4.3), and dynamically it can help find the first transformation that introduces a fault in a test program (Section 4.7).
- Hoopl implements subtle algorithms, including at least (a) interleaved analysis and rewriting, (b) speculative rewriting, (c) computing fixed points, and (d) dynamic fault isolation. Previous implementations of these algorithms—including three of our own—are complicated and hard to understand, because the tricky pieces are implemented all together, inseparably. A significant contribution of this paper is a new way to structure the implementation so that each tricky piece is handled in just one place, separate from all the others (Section 5). The result is sufficiently elegant that we emphasize the implementation as an object of interest in its own right.

A working prototype of Hoopl will soon be available for download at `http://ghc.cs.tufts.edu/hoopl`. It is no toy: an ancestor of this library is part of the Glasgow Haskell Compiler, where it optimizes the imperative C`--` code in GHC's back end. The new design is far nicer, and it will be in GHC shortly.

The API for Hoopl seems quite natural, but it requires relatively sophisticated aspects of Haskell's type system, such as higher-rank polymorphism, GADTs, and type functions. As such, Hoopl offers a compelling case study in the utility of these features.

2. Dataflow analysis & transformation by example

We begin by setting the scene, introducing some vocabulary, and showing a small motivating example. A control-flow graph, per-

[1] Using Hoopl, it is not necessary to have the more complex rule "if `x` is live after `x:=y+2` then `y` is live before it," because if `x` is *not* live after `x:=y+2`, the assignment `x:=y+2` will be eliminated.

haps representing the body of a procedure, is a collection of *basic blocks*—or just "blocks". Each block is a sequence of instructions, beginning with a label and ending with a control-transfer instruction that branches to other blocks. The goal of dataflow optimization is to compute valid *assertions* (or *dataflow facts*), then use those assertions to justify code-improving transformations (or *rewrites*) on a *control-flow graph*.

Consider a concrete example: constant propagation with constant folding. On the left we have a basic block; in the middle we have facts that hold between statements (or *nodes*) in the block; and at the right we have the result of transforming the block based on the assertions:

```
Before        Facts        After
    ------------{}-------------
x := 3+4                   x := 7
    ----------{x=7}------------
z := x>5                   z := True
    -------{x=7, z=True}-------
if z                       goto L1
 then goto L1
 else goto L2
```

Constant propagation works from top to bottom. We start with the empty fact. Given the empty fact and the node `x:=3+4` can we make a (constant-folding) transformation? Yes! We can replace the node with `x:=7`. Now, given this transformed node, and the original fact, what fact flows out of the bottom of the transformed node? The fact {`x=7`}. Given the fact {`x=7`} and the node `z:=x>5`, can we make a transformation? Yes: constant propagation can replace the node with `z:=7>5`. Now, can we do another transformation? Yes: constant folding can replace the node with `z:=True`. And so the process continues to the end of the block, where we can replace the conditional branch with an unconditional one, `goto L1`.

The example above is simple because the program has only straightline code; when programs have loops, dataflow analysis gets more complicated. For example, consider the following graph, where we assume `L1` is the entry point:

```
L1: x=3; y=4; if z then goto L2 else goto L3
L2: x=7; goto L3
L3: ...
```

Because control flows to `L3` from two places, we must *join* the facts coming from those two places. All paths to `L3` produce the fact `y=`4, so we can conclude that this fact holds at `L3`. But depending on the the path to `L3`, `x` may have different values, so we conclude "`x=`$\top$", meaning that there is no single value held by `x` at `L3`.[2] The final result of joining the dataflow facts that flow to `L3` is the new fact `x=`$\top$ $\wedge$ `y=4` $\wedge$ `z=`$\top$.

Interleaved transformation and analysis. Our example *interleaves* transformation and analysis. Interleaving makes it far easier to write effective analyses. If, instead, we *first* analyzed the block and *then* transformed it, the analysis would have to "predict" the transformations. For example, given the incoming fact {`x=7`} and the instruction `z:=x>5`, a pure analysis could produce the outgoing fact {`x=7, z=True`} by simplifying `x>5` to `True`. But the subsequent transformation must perform *exactly the same simplification* when it transforms the instruction to `z:=True`! If instead we *first* rewrite the node to `z:=True`, and *then* apply the transfer function to the new node, the transfer function becomes laughably simple: it merely has to see if the right hand side is a constant (you can see actual code in Section 4.6). The gain is even more compelling if there are a number of interacting analyses and/or transformations; for more substantial examples, consult Lerner, Grove, and Chambers (2002).

Forwards and backwards. Constant propagation works *forwards*, and a fact is typically an assertion about the program state (such as "variable `x` holds value `7`"). Some useful analyses work *backwards*. A prime example is live-variable analysis, where a fact takes the form "variable `x` is live" and is an assertion about the *continuation* of a program point. For example, the fact "`x` is live" at a program point P is an assertion that `x` is used on some program path starting at P. The accompanying transformation is called dead-code elimination; if `x` is not live, this transformation replaces the node `x:=e` with a no-op.

3. Representing control-flow graphs

Hoopl is a library that makes it easy to define dataflow analyses, and transformations driven by these analyses, on control-flow graphs. Graphs are composed from smaller units, which we discuss from the bottom up:

- A *node* is defined by Hoopl's client; Hoopl knows nothing about the representation of nodes (Section 3.2).
- A basic *block* is a sequence of nodes (Section 3.3).
- A *graph* is an arbitrarily complicated control-flow graph, composed from basic blocks (Section 3.4).

3.1 Shapes: Open and closed

Nodes, blocks, and graphs share important properties in common. In particular, each can be *open or closed at entry* and *open or closed at exit*. An *open* point is one at which control may implicitly "fall through;" to transfer control at a *closed* point requires an explicit control-transfer instruction. For example,

- A shift-left instruction is open on entry (because control can fall into it from the preceding instruction), and open on exit (because control falls through to the next instruction).
- An unconditional branch is open on entry, but closed on exit (because control cannot fall through to the next instruction).
- A label is closed on entry (because in Hoopl we do not allow control to fall through into a branch target), but open on exit.

These examples concern nodes, but the same classification applies to blocks and graphs. For example the block

```
x:=7; y:=x+2; goto L
```

is open on entry and closed on exit. This is the block's *shape*, which we often abbreviate "open/closed;" we may refer to an "open/closed block."

The shape of a thing determines that thing's control-flow properties. In particular, whenever E is a node, block, or graph,

- If E is open at the entry, it has a unique predecessor; if it is closed, it may have arbitrarily many predecessors—or none.
- If E is open at the exit, it has a unique successor; if it is closed, it may have arbitrarily many successors—or none.

3.2 Nodes

The primitive constituents of a Hoopl control-flow graph are *nodes*, which are defined by the client. Typically, a node might represent a machine instruction, such as an assignment, a call, or a conditional branch. But Hoopl's graph representation is polymorphic in the node type, so each client can define nodes as it likes. Because they contain nodes defined by the client, graphs can include arbitrary

[2] In this example `x` really does vary at `L3`, but in general the analysis might be conservative.

```
data Node e x where
  LabelNode  :: Label -> Node C O
  Assign     :: Var   -> Expr -> Node O O
  Store      :: Expr  -> Expr -> Node O O
  Branch     :: Label -> Node O C
  CondBranch :: Expr  -> Label -> Label -> Node O C
    -- ... more constructors ...
```

Figure 1. A typical node type as it might be defined by a client

client-specified data, including (say) C statements, method calls in an object-oriented language, or whatever.

Hoopl knows *at compile time* whether a node is open or closed at entry and exit: the type of a node has kind `*->*->*`, where the two type parameters are type-level flags, one for entry and one for exit. Such a type parameter may be instantiated only with type `O` (for open) or type `C` (for closed). As an example, Figure 1 shows a typical node type as it might be written by one of Hoopl's clients. The type parameters are written `e` and `x`, for entry and exit respectively. The type is a generalized algebraic data type; the syntax gives the type of each constructor. For example, constructor `LabelNode` takes a `Label` and returns a node of type `Node C O`, where the "`C`" says "closed at entry" and the "`O`" says "open at exit". The types `Label`, `O`, and `C` are defined by Hoopl (Figure 2).

Similarly, an `Assign` node takes a variable and an expression, and returns a `Node` open at both entry and exit; the `Store` node is similar. The types `Var` and `Expr` are private to the client, and Hoopl knows nothing of them. Finally, the control-transfer nodes `Branch` and `CondBranch` are open at entry and closed at exit.

Nodes closed on entry are the only targets of control transfers; nodes open on entry and exit never perform control transfers; and nodes closed on exit always perform control transfers[3]. Because of the position each type of node occupies in a basic block, we often call them *first*, *middle*, and *last* nodes respectively.

3.3 Blocks

Hoopl combines the client's nodes into blocks and graphs, which, unlike the nodes, are defined by Hoopl (Figure 2). A `Block` is parameterized over the node type `n` as well as over the same flag types that make it open or closed at entry and exit.

The `BUnit` constructor lifts a node to become a block; `BCat` concatenates blocks in sequence. It makes sense to concatenate blocks only when control can fall through from the first to the second; therefore, two blocks may be concatenated only if each block is open at the point of concatenation. This restriction is enforced by the type of `BCat`, whose first argument must be open on exit, and whose second argument must be open on entry. It is statically impossible, for example, to concatenate a `Branch` immediately before an `Assign`. Indeed, the `Block` type statically guarantees that any closed/closed `Block`—which compiler writers normally call a "basic block"—consists of exactly one closed/open node (such as `Label` in Figure 1), followed by zero or more open/open nodes (`Assign` or `Store`), and terminated with exactly one open/closed node (`Branch` or `CondBranch`). Using GADTs to enforce these invariants is one of Hoopl's innovations.

[3] To obey these invariants, a node for a conditional-branch instruction, which typically either transfers control *or* falls through, must be represented as a two-target conditional branch, with the fall-through path in a separate block. This representation is standard (Appel 1998), and it costs nothing in practice: such code is easily sequentialized without superfluous branches.

```
data O   -- Open
data C   -- Closed

data Block n e x where
 BUnit :: n e x       -> Block n e x
 BCat  :: Block n e O -> Block n O x -> Block n e x

data Graph n e x where
  GNil  :: Graph n O O
  GUnit :: Block n O O -> Graph n O O
  GMany :: MaybeO e (Block n O C)
        -> Body n
        -> MaybeO x (Block n C O)
        -> Graph n e x

data Body n where
  BodyEmpty :: Body n
  BodyUnit  :: Block n C C -> Body n
  BodyCat   :: Body n -> Body n -> Body n

data MaybeO ex t where
  JustO    :: t -> MaybeO O t
  NothingO ::      MaybeO C t

newtype Label = Label Int

class Edges n where
  entryLabel :: n C x -> Label
  successors :: n e C -> [Label]
```

Figure 2. The block and graph types defined by Hoopl

3.4 Graphs

Hoopl composes blocks into graphs, which are also defined in Figure 2. Like `Block`, the data type `Graph` is parameterized over both nodes `n` and its open/closed shape (`e` and `x`). It has three constructors. The first two deal with the base cases of open/open graphs: an empty graph is represented by `GNil` while a single-block graph is represented by `GUnit`.

More general graphs are represented by `GMany`, which has three fields: an optional entry sequence, a body, and an optional exit sequence.

- If the graph is open at the entry, it contains an entry sequence of type `Block n O C`. We could represent this sequence as a value of type `Maybe (Block n O C)`, but we can do better: a value of `Maybe` type requires a *dynamic* test, but we know *statically*, at compile time, that the sequence is present if and only if the graph is open at the entry. We express our compile-time knowledge by using the type `MaybeO e (Block n O C)`, a type-indexed version of `Maybe` which is also defined in Figure 2: the type `MaybeO O a` is isomorphic to `a`, while the type `MaybeO C a` is isomorphic to `()`.
- The body of the graph is a collection of closed/closed blocks. To be able to concatenate bodies in constant time, we introduce the representation `Body n`.
- The exit sequence is dual to the entry sequence, and like the entry sequence, its presence or absence is deducible from the static type of the graph.

Graphs concatenate nicely, in constant time. Unlike blocks, two graphs may be concatenated not only when they are both open at

```
data FwdPass n f
  = FwdPass { fp_lattice  :: DataflowLattice f
            , fp_transfer :: FwdTransfer n f
            , fp_rewrite  :: FwdRewrite n f }

------- Lattice ----------
data DataflowLattice a = DataflowLattice
 { fact_bot    :: a
 , fact_extend :: a -> a -> (ChangeFlag, a) }

data ChangeFlag = NoChange | SomeChange

------- Transfers ----------
type FwdTransfer n f
  = forall e x. n e x -> Fact e f -> Fact x f

------- Rewrites ----------
type FwdRewrite n f
  = forall e x. n e x -> Fact e f
           -> Maybe (FwdRes n f e x)

data FwdRes n f e x
  = FwdRes (AGraph n e x) (FwdRewrite n f)

------- Fact-like things -------
type family   Fact x f :: *
type instance Fact O f = f
type instance Fact C f = FactBase f

------- FactBase -------
type FactBase f = LabelMap f
 -- A finite mapping from Labels to facts f
```

Figure 3. Hoopl API data types

the point of concatenation but also when they are both closed—and not in the other two cases:

```
gCat :: Graph n e a -> Graph n a x -> Graph n e x
gCat GNil g2 = g2
gCat g1 GNil = g1

gCat (GUnit b1) (GUnit b2) = GUnit (b1 `BCat` b2)

gCat (GUnit b) (GMany (JustO e) bs x)
  = GMany (JustO (b `BCat` e)) bs x

gCat (GMany e bs (JustO x)) (GUnit b2)
  = GMany e bs (JustO (x `BCat` b2))

gCat (GMany e1 bs1 (JustO x1)) (GMany (JustO e2) bs2 x2)
  = GMany e1 (bs1 `BodyCat` b `BodyCat` bs2) x2
  where b = BodyUnit (x1 `BCat` e2)

gCat (GMany e1 bs1 NothingO) (GMany NothingO bs2 x2)
  = GMany e1 (bs1 `BodyCat` bs2) x2
```

This definition illustrates the power of GADTs: the pattern matching is exhaustive, and all the open/closed invariants are statically checked. For example, consider the second-last equation for `gCat`. Since the exit link of the first argument is `JustO x1`, we know that type parameter `a` is `O`, and hence the entry link of the second argument must be `JustO e2`. Moreover, block `x1` must be closed/open, and block `e2` must be open/closed. We can therefore concatenate them with `BCat` to produce a closed/closed block, which is added to the `Body` of the result.

We have carefully crafted the types so that if `BCat` and `BodyCat` are considered as associative operators, every graph has a unique representation. To guarantee uniqueness, `GUnit` is restricted to open/open blocks. If `GUnit` were more accommodating, there would be more than one way to represent some graphs, and it wouldn't be obvious to a client which representation to choose—or if the choice made a difference.

3.5 Labels and successors

If Hoopl knows nothing about nodes, how can it know where a control transfer goes, or what is the `Label` at the start of a block? To answer such questions, the standard Haskell idiom is to define a type class whose methods provide exactly the operations needed; Hoopl's type class, called `Edges`, is given in Figure 2. The `entryLabel` method takes a first node (one closed on entry, Section 3.2) and returns its `Label`; the `successors` method takes a last node (closed on exit) and returns the `Label`s to which it can transfer control. A middle node, which is open at both entry and exit, cannot refer to any `Label`s, so no corresponding interrogation function is needed.

A node type defined by a client must be an instance of `Edges`. In Figure 1, the client's instance declaration for `Node` would be

```
instance Edges Node where
  entryLabel (LabelNode l) = l
  successors (Branch b) = [b]
  successors (CondBranch e b1 b2) = [b1,b2]
```

Again, the pattern matching for both functions is exhaustive, and the compiler statically checks this fact. Here, `entryLabel` cannot be applied to an `Assign` or `Branch` node, and any attempt to define a case for `Assign` or `Branch` would result in a type error.

While it is required for the client to provide this information about nodes, it is very convenient for Hoopl to get the same information about blocks. For its own internal use, Hoopl provides this instance declaration for the `Block` type:

```
instance Edges n => Edges (Block n) where
  entryLabel (BUnit n)  = entryLabel n
  entryLabel (BCat b _) = entryLabel b
  successors (BUnit n)  = successors n
  successors (BCat _ b) = successors b
```

Because the functions `entryLabel` and `successors` are used to track control flow *within* a graph, Hoopl does not need to ask for the entry label or successors of a `Graph` itself. Indeed, `Graph` *cannot* be an instance of `Edges`, because even if a `Graph` is closed at the entry, it does not have a unique entry label.

4. Using Hoopl to analyze and transform graphs

Now that we have graphs, how do we optimize them? Hoopl makes it easy for a client to build a new dataflow analysis and optimization. The client must supply the following pieces:

- *A node type* (Section 3.2). Hoopl supplies the `Block` and `Graph` types that let the client build control-flow graphs out of nodes.
- *A data type of facts* and some operations over those facts (Section 4.1). Each analysis uses facts that are specific to that particular analysis, which Hoopl accommodates by being polymorphic in the fact type.
- *A transfer function* that takes a node and returns a *fact transformer*, which takes a fact flowing into the node and returns the transformed fact that flows out of the node (Section 4.2).
- *A rewrite function* that takes a node and an input fact, and which returns either `Nothing` or `(Just g)` where `g` is a graph

Part of optimizer	*Specified by*	*Implemented by*	*How many*
Control-flow graphs	Us	Us	One
Nodes in a control-flow graph	You	You	One type per intermediate language
Dataflow fact F	You	You	One type per logic
Lattice operations	Us	You	One set per logic
Transfer functions	Us	You	One per analysis
Rewrite functions	Us	You	One per transformation
Solve-and-rewrite functions	Us	Us	Two (forward, backward)

Table 4. Parts of an optimizer built with Hoopl

that should replace the node. The ability to replace a *node* by a *graph* that may include internal control flow is crucial for many code-improving transformations. We discuss the rewrite function in Sections 4.3 and 4.4.

These requirements are summarized in Table 4. Because facts, transfer functions, and rewrite functions work closely together, we represent their combination as a single record of type `FwdPass` (Figure 3). The elements of `FwdPass` are, and must be, polymorphic functions—Hoopl must use higher-rank types.

Given a node type `n` and a `FwdPass`, a client can ask Hoopl to analyze and rewrite a closed/closed graph represented as `Body n`:

```
analyzeAndRewriteFwd
  :: Edges n            -- Access to flow edges
  => FwdPass n f   -- Lattice, transfer,
                        -- and rewrite functions
  -> Body n                  -- Input body
  -> FactBase f              -- Input fact(s)
  -> FuelMonad (Body n,      -- Result body
               FactBase f) -- ...and its facts
```

Given a `FwdPass`, the analyze-and-rewrite function transforms a `Body` into an optimized `Body`. As its type shows, this function is polymorphic in the types of nodes `n` and facts `f`; these types are determined entirely by the client.

As well as taking and returning a `Body`, the function also takes input facts (the `FactBase`) and produces output facts. A `FactBase` is simply a finite mapping from `Label` to facts. The output `FactBase` maps each `Label` in the `Body` to its fact; if the `Label` is not in the domain of the `FactBase`, its fact is the bottom element of the lattice. Similarly the input `FactBase` supplies any facts that hold on entry to the `Body`. For example, in our constant-propagation example from Section 2, if the `Body` represents the body of a procedure with parameters x, y, z, we would map the entry `Label` to a fact `x=`$\top$ $\wedge$ `y=`$\top$ $\wedge$ `z=`$\top$, to specify that the procedure's parameters may not be constants.

The client's model of how `analyzeAndRewriteFwd` works is as follows: Hoopl walks forward over each block in the graph. At each node, Hoopl applies the rewrite function to the node and the incoming fact. If the rewrite function returns `Nothing`, the node is retained as part of the output graph, the transfer function is used to compute the downstream fact, and Hoopl moves on to the next node. But if the rewrite function returns `(Just g)`, indicating that it wants to rewrite the node to the replacement graph `g`, then Hoopl recursively analyzes and rewrites `g` before moving on to the next node. A node following a rewritten node sees *up-to-date* facts; that is, its input fact is computed by analyzing the replacement graph.

In this section we flesh out the *interface* to `analyzeAndRewriteFwd`, leaving the implementation for Section 5.

4.1 Dataflow lattices

For each analysis or transformation, the client must define a type of dataflow facts. A dataflow fact often represents an assertion about a program point,[4] but in general, dataflow analysis establishes properties of *paths*:

- An assertion about all paths *to* a program point is established by a *forwards analysis*. For example the assertion "`x` $=$ `3`" at point P claims that variable `x` holds value `3` at P, regardless of the path by which P is reached.
- An assertion about all paths *from* a program point is established by a *backwards analysis*. For example, the assertion "`x` is dead" at point P claims that no path from P uses variable `x`.

A set of dataflow facts must form a lattice, and Hoopl must know (a) the bottom element of the lattice and (b) how to take the least upper bound (join) of two elements. To ensure that analysis terminates, it is enough if every fact has a finite number of distinct facts above it, so that repeated joins eventually reach a fixed point.

In practice, joins are computed 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.

The bottom element and join operation of a lattice of facts of type `f` are stored in a value of type `DataflowLattice f` (Figure 3). As noted in the previous paragraph, Hoopl needs to know when the result of a join is equal to one of the arguments joined. Because this information is often available very cheaply at the time when the join is computed, Hoopl does not require a separate equality test on facts (which might be expensive). Instead, Hoopl requires that `fact_extend` return a `ChangeFlag` as well as the least upper bound. The `ChangeFlag` should be `NoChange` if the result is the same as the first argument (the old fact), and `SomeChange` if the result differs. (Function `fact_extend` is *not* symmetric in its arguments.)

4.2 The transfer function

A forward transfer function is presented with the dataflow fact(s) on the edge(s) coming into a node, and it computes dataflow fact(s) on the outgoing edge(s). In a forward analysis, the dataflow engine starts with the fact at the beginning of a block and applies the transfer function to successive 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, consider this graph, with entry at `L1`:

```
L1: x=3; goto L2
L2: y=x+4; x=x-1;
    if x>0 then goto L2 else return
```

A forward analysis starts with the bottom fact {} at every label. Analyzing `L1` propagates this fact forward, by applying the transfer function successively to the nodes of `L1`, emerging with the fact {`x=3`} for `L2`. This new fact is joined with the existing (bottom) fact for `L2`. Now the analysis propagates `L2`'s fact forward, again using the transfer function, this time emerging with a new fact {`x=2, y=7`} for `L2`. Again, the new fact is joined with the existing fact for `L2`, and the process is iterated until the facts for each label reach a fixed point.

But wait! What is the *type* of the transfer function? If the node is open at exit, the transfer function produces a single fact. But what

[4] In Hoopl, a program point is simply an edge in a control-flow graph.

```
type AGraph n e x
  = [Label] -> (Graph n e x, [Label])

withLabels :: Int -> ([Label] -> AGraph n e x)
           -> AGraph n e x
withLabels n fn = \ls -> fn (take n ls) (drop n ls)

mkIfThenElse :: Expr -> AGraph Node O O
             -> AGraph Node O O -> AGraph Node O O
mkIfThenElse p t e
  = withLabels 3 $ \[l1,l2,l3] ->
    gUnitOC (BUnit (CondBranch p l1 l2))   `gCat`
    mkLabel l1 `gCat` t `gCat` mkBranch l3 `gCat`
    mkLabel l2 `gCat` e `gCat` mkBranch l3 `gCat`
    mkLabel l3

mkLabel  l = gUnitCO (BUnit (LabelNode l))
mkBranch l = gUnitOC (BUnit (Branch l))
gUnitOC  b = GMany (JustO b) BodyEmpty   NothingO
gUnitCO  b = GMany NothingO  BodyEmpty   (JustO b)
```

Figure 5. The `AGraph` type and example constructions

if the node is *closed* on exit? In that case the transfer function must produce a list of (`Label`,fact) pairs, one for each outgoing edge. *So the type of the transfer function's result depends on the shape of the node's exit.* Fortunately, this dependency can be expressed precisely, at compile time, by Haskell's (recently added) *indexed type families*. The relevant Hoopl definitions are given in Figure 3. A forward transfer function, of type (`FwdTransfer n f`), is a function polymorphic in `e` and `x`. It takes a node of type (`n e x`) and a fact of type `f`, and it produces an outgoing "fact-like thing" of type (`Fact x f`). The type constructor `Fact` should be thought of as a type-level function; its signature is given in the `type family` declaration, while its definition is given by two `type instance` declarations. The first declaration says that the fact-like thing coming out of a node *open* at the exit is just a fact `f`. The second declaration says that the fact-like thing coming out of a node *closed* at the exit is a mapping from `Label` to facts.

We have ordered the arguments such that if

```
  transfer_fn :: FwdTransfer n f
  node        :: n e x
```

then `(transfer_fn node)` is a predicate transformer:

```
  transfer_fn node :: Fact e f -> Fact x f
```

4.3 The rewrite function

We compute dataflow facts in order to enable code-improving transformations. In our constant-propagation example, the dataflow facts may enable us to simplify an expression by performing constant folding, or to turn a conditional branch into an unconditional one. Similarly, a liveness analysis may allow us to replace a dead assignment with a no-op.

A `FwdPass` therefore includes a *rewriting function*, whose type, `FwdRewrite`, is given in Figure 3. A rewriting function takes a node and a fact, and optionally returns. . . what? At first one might expect that rewriting should return a new node, but that is not enough: We might want to remove a node by rewriting it to the empty graph, or more ambitiously, we might want to replace a high-level operation with a tree of conditional branches or a loop, which would entail introducing new blocks with internal control flow. In general, a rewrite function must be able to return a *graph*.

Concretely, a `FwdRewrite` takes a node and a suitably shaped fact, and returns either `Nothing`, indicating that the node should not be replaced, or `(Just (FwdRes` g rw`))`, indicating that the node should be replaced with g: the replacement graph. You may have been expecting g to have type `Graph n e x`, but it actually has type `AGraph n e x`. The reason is that if the rewriter makes graphs containing blocks, it may need fresh `Label`s. An `AGraph` provides easy access to fresh labels, using `withLabels` (Figure 5). The figure also shows an implementation of `AGraph` and a few simple functions typically used to build `AGraph`s.

The type of `FwdRewrite` in Figure 3 guarantees *at compile time* that the replacement graph g has the *same* open/closed shape as the node being rewritten. For example, a branch instruction can be replaced only by a graph closed at the exit. Moreover, because only an open/open graph can be empty—look at the type of `GNil` in Figure 2—the type of `FwdRewrite` guarantees, at compile time, that no head of a block (closed/open) or tail of a block (open/closed) can ever be deleted by being rewritten to an empty graph.

4.4 Shallow vs deep rewriting

Once the rewrite has been performed, what then? Since the rewrite returns a graph, the replacement graph must itself be analyzed, and its nodes may be rewritten. So we must call `analyzeAndRewriteFwd` to process the replacement graph—but what `FwdPass` should we use? There are two common situations:

- Sometimes we want to analyze and transform the replacement graph with an unmodified `FwdPass`, further rewriting the replacement graph. This procedure is called *deep rewriting*. When deep rewriting is used, the client's rewrite function must ensure that the graphs it produces are not rewritten indefinitely (Section 4.9).
- Sometimes we want to analyze *but not further rewrite* the replacement graph. This procedure is called *shallow rewriting*. It is easily implemented by using a modified `FwdPass` whose rewriting function always returns `Nothing`.

Deep rewriting is essential to achieve the full benefits of interleaved analysis and transformation (Lerner, Grove, and Chambers 2002). But shallow rewriting can be vital as well; for example, a forward dataflow pass that inserts a spill before a call must not rewrite the call again, lest it attempt to insert infinitely many spills.

An innovation of Hoopl is to build the choice of shallow or deep rewriting into each rewrite function, an idea that is elegantly captured by the `FwdRes` type returned by a `FwdRewrite` (Figure 3). The first component of the `FwdRes` is the replacement graph, as discussed earlier. The second component, rw, is a *new rewriting function* to use when recursively processing the replacement graph. For shallow rewriting this new function is the constant `Nothing` function; for deep rewriting it is the original rewriting function.

4.5 Composing rewrite functions and dataflow passes

By requiring each rewrite to return a new rewrite function, Hoopl enables a variety of combinators over rewrite functions. For example, here is a function that combines two rewriting functions in sequence:

```
thenFwdRw :: FwdRewrite n f
          -> FwdRewrite n f
          -> FwdRewrite n f
thenFwdRw rw1 rw2 n f
  = case rw1 n f of
      Nothing               -> rw2 n f
      Just (FwdRes g rw1a) -> Just $ FwdRes g $
                               rw1a `thenFwdRw` rw2
  noFwdRw :: FwdRewrite n f
  noFwdRw n f = Nothing
```

What a beautiful type `thenFwdRw` has! It tries `rw1`, and if `rw1` declines to rewrite, it behaves like `rw2`. But if `rw1` rewrites, returning a new rewriter `rw1a`, then the overall call also succeeds, returning a new rewrite function obtained by combining `rw1a` with `rw2`. (We cannot apply `rw1a` or `rw2` directly to the replacement graph `g`, because `r1` returns a graph and `rw2` expects a node.) The rewriter `noFwdRw` is the identity of `thenFwdRw`. Finally, `thenFwdRw` can combine a deep-rewriting function and a shallow-rewriting function, to produce a rewriting function that is a combination of deep and shallow.

A shallow rewriting function can be made deep by iterating it:

```
iterFwdRw :: FwdRewrite n f -> FwdRewrite n f
iterFwdRw rw =
 \n f -> case rw n f of
          Just (FwdRes g rw2) ->
            Just $ FwdRes g (rw2 ‘thenFwdRw‘ iterFwdRw rw)
          Nothing -> Nothing
```

If we have shallow rewrites A and B then we can build AB, A^*B, $(AB)^*$, and so on: sequential composition is `thenFwdRw` and the Kleene star is `iterFwdRw`.◇

NR: Do we still believe this claim?

The combinators above operate on rewrite functions that share a common fact type and transfer function. It can also be useful to combine entire dataflow passes that use different facts. We invite you to write one such combinator, with type

```
  thenFwd :: FwdPass n f1
          -> FwdPass n f2
          -> FwdPass n (f1,f2)
```

The two passes run interleaved, not sequentially, and each may help the other, yielding better results than running A and then B or B and then A (Lerner, Grove, and Chambers 2002).

4.6 Example: Constant propagation and constant folding

Figure 6 shows client code for constant propagation and constant folding. For each variable at each point in a graph, the analysis concludes one of three facts: the variable holds a constant value (Boolean or integer), the variable might hold a non-constant value, or nothing is known about what the variable holds. We represent these facts using a finite map from a variable to a fact of type (`Maybe HasConst`). A variable with a constant value maps to `Just k`, where `k` is the constant value; a variable with a non-constant value maps to `Just Top`; and a variable with an unknown value maps to `Nothing` (i.e., it is not in the domain of the finite map).

The definition of the lattice (`constLattice`) is straightforward. The bottom element is an empty map (nothing is known about the contents of any variable). We use the `stdMapJoin` function to lift the join operation for a single variable (`constFactAdd`) up to the map containing facts for all variables.

For the transfer function, `varHasConst`, there are two interesting kinds of nodes: assignment and conditional branch. In the first two cases for assignment, a variable gets a constant value, so we produce a dataflow fact mapping the variable to its value. In the third case for assignment, the variable gets a non-constant value, so we produce a dataflow fact mapping the variable to `Top`. The last interesting case is a conditional branch where the condition is a variable. If the conditional branch flows to the true successor, the variable holds `True`, and similarly for the false successor. We update the fact flowing to each successor accordingly.

```
-- Types and definition of the lattice
data HasConst = Top | B Bool | I Integer
type ConstFact = Map.Map Var HasConst
constLattice = DataflowLattice
  { fact_bot    = Map.empty
  , fact_extend = stdMapJoin constFactAdd }
  where
    constFactAdd old new = (c, j)
      where j = if new == old then new else Top
            c = if j == old then NoChange else SomeChange

-------------------------------------------------------
-- Analysis: variable has constant value
varHasConst :: FwdTransfer Node ConstFact
varHasConst (LabelNode l)       f = lookupFact f l
varHasConst (Store _ _)         f = f
varHasConst (Assign x (Bool b)) f = Map.insert x (B b) f
varHasConst (Assign x (Int  i)) f = Map.insert x (I i) f
varHasConst (Assign x _)        f = Map.insert x Top   f
varHasConst (Branch l)          f = mkFactBase [(l, f)]
varHasConst (CondBranch (Var x) tid fid) f
  = mkFactBase [(tid, Map.insert x (B True)  f),
                (fid, Map.insert x (B False) f)]
varHasConst (CondBranch _ tid fid) f
  = mkFactBase [(tid, f), (fid, f)]

-------------------------------------------------------
-- Constant propagation
constProp :: FwdRewrite Node ConstFact
constProp node facts
  = fmap toAGraph (mapE rewriteE node)
  where
    rewriteE e (Var x)
      = case M.lookup x facts of
          Just (B b) -> Just $ Bool b
          Just (I i) -> Just $ Int  i
          _          -> Nothing
    rewriteE e = Nothing

-------------------------------------------------------
-- Simplification ("constant folding")
simplify :: FwdRewrite Node f
simplify (CondBranch (Bool b) t f) _
  = Just $ toAGraph $ Branch (if b then t else f)
simplify node _ = fmap toAGraph (mapE s_exp node)
  where
    s_exp (Binop Add (Int i1) (Int i2))
       = Just $ Int $ i1 + i2
    ...  -- more cases for constant folding

-- Rewriting expressions
mapE :: (Expr    -> Maybe Expr)
      -> Node e x -> Maybe (Node e x)
mapE f (LabelNode _) = Nothing
mapE f (Assign x e)  = fmap (Assign x) $ f e
 ...  -- more cases for rewriting expressions

-------------------------------------------------------
-- Defining the forward dataflow pass
constPropPass = FwdPass
   { fp_lattice  = constLattice
   , fp_transfer = varHasConst
   , fp_rewrite  = constProp ‘thenFwdRw‘ simplify }
```

Figure 6. The client for constant propagation and constant folding

We do not need to consider complicated cases such as an assignment `x:=y` where `y` holds a constant value `k`. Instead, we rely on the interleaving of transformation and analysis to first transform the assignment to `x:=k`, which is exactly what our simple transfer function expects. As we mention in Section 2, interleaving makes it possible to write the simplest imaginable transfer functions, without missing opportunities to improve the code.

The rewrite function for constant propagation, `constProp`, simply rewrites each use of a variable to its constant value. We use the auxiliary function `mapE` to apply `rewriteE` to each use of a variable in each kind of node; in turn, the `rewriteE` function checks if the variable has a constant value and makes the substitution. We assume an auxiliary function

```
  toAGraph :: Node e x -> AGraph e x
```

Figure 6 also gives a completely separate rewrite function to perform constant folding, called `simplify`. It rewrites a conditional branch on a boolean constant to an unconditional branch, and to find constant subexpressions, it runs `s_exp` on every subexpression. Function `simplify` does not need to check whether a variable holds a constant value; it relies on `constProp` to have replaced the variable by the constant. Indeed, `simplify` does not consult the incoming fact at all, and hence is polymorphic in `f`.

We have written two `FwdRewrite` functions because they are independently useful. But in this case we want to apply *both* of them, so we compose them with `thenFwdRw`. The composed rewrite functions, along with the lattice and the transfer function, go into `constPropPass` (bottom of Figure 6). To improve a particular graph, we pass `constPropPass` and the graph to `analyzeAndRewriteFwd`.

4.7 Throttling the dataflow engine using "optimization fuel"

Debugging an optimization can be tricky: an optimization may rewrite hundreds of nodes, and any of those rewrites could be incorrect. To debug dataflow optimizations, we use Whalley's (1994) powerful technique to identify the first rewrite that transforms a program from working code to faulty code.

The key idea is to limit the number of rewrites that are performed while optimizing a graph. In Hoopl, the limit is called *optimization fuel*: each rewrite costs one unit of fuel, and when the fuel is exhausted, no more rewrites are permitted. Because each rewrite leaves the observable behavior of the program unchanged, it is safe to stop rewriting at any point. Given a program that fails when compiled with optimization, a test infrastructure uses binary search on the amount of optimization fuel, until it finds that the program works correctly after $n - 1$ rewrites but fails after n rewrites. The nth rewrite is faulty.

You may have noticed that `analyzeAndRewriteFwd` returns a value in the `FuelMonad` (Section 4). The `FuelMonad` is a simple state monad maintaining the supply of unused fuel. It also holds a supply of fresh labels, which are used by the rewriter for making new blocks; more precisely, Hoopl uses these labels to take the `AGraph` returned by a pass's rewrite function (Figure 3) and convert it to a `Graph`.

4.8 Fixed points and speculative rewrites

Are rewrites sound, especially when there are loops? Many analyses compute a fixed point starting from unsound "facts"; for example, a live-variable analysis starts from the assumption that all variables are dead. This means *rewrites performed before a fixed point is reached may be unsound, and their results must be discarded*. Each iteration of the fixed-point computation must start afresh with the original graph.

Although the rewrites may be unsound, *they must be performed* (speculatively, and possibly recursively), so that the facts downstream of the replacement graphs are as accurate as possible. For example, consider this graph, with entry at `L1`:

```
  L1: x=0; goto L2
  L2: x=x+1; if x==10 then goto L3 else goto L2
```

The first traversal of block `L2` starts with the unsound "fact" $\{x=0\}$; but analysis of the block propagates the new fact $\{x=1\}$ to `L2`, which joins the existing fact to get $\{x=\top\}$. What if the predicate in the conditional branch were `x<10` instead of `x==10`? Again the first iteration would begin with the tentative fact $\{x=0\}$. Using that fact, we would rewrite the conditional branch to an unconditional branch `goto L3`. No new fact would propagate to `L2`, and we would have successfully (and soundly) eliminated the loop. This example is contrived, but it illustrates that for best results we should

- Perform the rewrites on every iteration.
- Begin each new iteration with the original, virgin graph.

This sort of algorithm is hard to implement in an imperative setting, where rewrites mutate a graph in place. But with an immutable graph, implementing the algorithm is trivially easy: we simply revert to the original graph at the start of each fixed-point iteration.

4.9 Correctness

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

Hoopl requires that a client's functions meet these preconditions:

- The lattice must have no *infinite ascending chains*; that is, every sequence of calls to `fact_extend` must eventually return `NoChange`.
- The transfer function must be *monotonic*: given a more informative fact in, it should produce a more informative fact out.
- The rewrite function must be *sound*: if it replaces a node `n` by a replacement graph `g`, then `g` must be observationally equivalent to `n` under the assumptions expressed by the incoming dataflow fact `f`.
- The rewrite function must be *consistent* with the transfer function; that is, `transfer n f` $\sqsubseteq$ `transfer g f`. For example, if the analysis says that `x` is dead before the node `n`, then it had better still be dead if `n` is replaced by `g`.
- To ensure termination, a transformation that uses deep rewriting must not return replacement graphs which contain nodes that could be rewritten indefinitely.

Without the conditions on monotonicity and consistency, our algorithm will terminate, but there is no guarantee that it will compute a fixed point of the analysis. And that in turn threatens the soundness of rewrites based on possibly bogus "facts".

However, when the preconditions above are met,

- The algorithm terminates. The fixed-point loop must terminate because the lattice has no infinite ascending chains. And the client is responsible for avoiding infinite recursion when deep rewriting is used.
- The algorithm is sound. Why? Because if each rewrite is sound (in the sense given above), then applying a succession of rewrites is also sound. Moreover, a sound analysis of the replacement graph may generate only dataflow facts that could have been generated by a more complicated analysis of the original graph.

```
data RG n f e x where
  RGNil   :: RG n f a a
  RGCatO  :: RG n f e O -> RG n f O x -> RG n f e x
  RGCatC  :: RG n f e C -> RG n f C x -> RG n f e x
  RGUnit  :: Fact e f  -> Block n e x -> RG n f e x
```

Figure 7. The data type RG of rewritten graphs

5. Hoopl's implementation

Section 4 gives a client's-eye view of Hoopl, showing how to use it to create analyses and transformations. Hoopl's interface is simple, but the *implementation* of interleaved analysis and rewriting is quite complicated. Lerner, Grove, and Chambers (2002) do not describe their implementation. We have written at least three previous implementations, all of which were long and hard to understand, and only one of which provided compile-time guarantees about open and closed shapes. We are not confident that any of these implementations are correct.

In this paper we describe our new implementation. It is short (about a third of the size of our last attempt), elegant, and offers strong static shape guarantees. The whole thing is about 300 lines of code, excluding comments; this count includes both forward and backward dataflow analysis and transformation.

We describe the implementation of *forward* analysis and transformation. The implementations of backward analysis and transformation are exactly analogous and are included in Hoopl.

5.1 Overview

We concentrate on implementing `analyzeAndRewriteFwd`, whose type is in Section 4. Its implementation is built on the hierarchy of nodes, blocks, and graphs described in Section 3. For each thing in the hierarchy, we develop a function of this type:

```
type ARF thing n
 = forall f e x. FwdPass n f
              -> thing e x -> Fact e f
              -> FuelMonad (RG n e x, Fact x f)
```

An ARF (short for "analyze and rewrite forward") is a combination of a rewrite and transfer function. An ARF takes a FwdPass, a thing (a node, block, or graph), and an input fact, and it returns a rewritten graph of type (RG n e x) of the same shape as the thing, plus a suitably shaped output fact. The type RG is internal to Hoopl; it is not seen by any client. We use it, not Graph, for two reasons:

- The client is often interested not only in the facts flowing out of the graph (which are returned in the Fact x f), but also in the facts on the *internal* blocks of the graph. A replacement graph of type (RG n e x) is decorated with these internal facts.
- A Graph has deliberately restrictive invariants; for example, a GMany with a JustO is always open at exit (Figure 2). It turns out to be awkward to maintain these invariants *during* rewriting, but easy to restore them *after* rewriting by "normalizing" an RG.

The information in an RG is returned to the client by the normalization function normalizeBody, which splits an RG into a Body and its corresponding FactBase:

```
normalizeBody :: Edges n => RG n f C C
              -> (Body n, FactBase f)
```

The constructors of RG are given in Figure 7. The essential points are that constructor RGUnit is polymorphic in the shape of a block, RGUnit carries a fact as well as a block, and the concatenation constructors record the shapes of the graphs at the point of concatenation. (A record of the shapes is needed so that when normalizeBody is presented with a block carried by RGUnit, it is known whether the block is an entry sequence, an exit sequence, or a basic block.)◇

NR: Within Hoopl, the RG *type is a great convenience. Mutter mutter: it carries facts as well as blocks, and it frees the client's rewrite functions from any obligation to respect the invariants of type* Graph*—I'm not convinced.*

We exploit the type distinctions of nodes, Block, Body, and Graph to structure the code into several small pieces, each of which can be understood independently. Specifically, we define a layered set of functions, each of which calls the previous one:

```
arfNode  :: Edges n => ARF n n
arfBlock :: Edges n => ARF (Block n) n
arfBody  :: Edges n
         => FwdPass n f -> Body n -> FactBase f
         -> FuelMonad (RG n f C C, FactBase f)
arfGraph :: Edges n => ARF (Graph n) n
```

- The arfNode function processes nodes (Section 5.3). It handles the subtleties of interleaved analysis and rewriting, and it deals with fuel consumption. It calls arfGraph to analyze and transform rewritten graphs.
- Based on arfNode it is extremely easy to write arfBlock, which lifts the analysis and rewriting from nodes to blocks (Section 5.2).
- Using arfBlock we define arfBody, which analyzes and rewrites a Body: that is, a group of closed/closed blocks linked by arbitrary control flow. Because a Body is always closed/closed and does not take shape parameters, function arfBody is less polymorphic than the others, but its type is what would be obtained by expanding and specializing the definition of ARF for a thing which is always closed/closed and is equivalent to a Body.

  Function arfBody takes care of fixed points (Section 5.4).
- Based on arfBody it is easy to write arfGraph (Section 5.2).

Given these functions, writing the main analyzer is a simple matter of matching the external API to the internal functions:

```
analyzeAndRewriteFwd
   :: forall n f. Edges n
   => FwdPass n f -> Body n -> FactBase f
   -> FuelMonad (Body n, FactBase f)

analyzeAndRewriteFwd pass body facts
  = do { (rg, _) <- arfBody pass body facts
       ; return (normalizeBody rg) }
```

5.2 From nodes to blocks

We begin our explanation with the second task: writing arfBlock, which analyzes and transforms blocks.

```
arfBlock :: Edges n => ARF (Block n) n
arfBlock pass (BUnit node) f
  = arfNode pass node f
arfBlock pass (BCat b1 b2) f
  = do { (g1,f1) <- arfBlock pass b1 f
       ; (g2,f2) <- arfBlock pass b2 f1
       ; return (g1 `RGCatO` g2, f2) }
```

The code is delightfully simple. The BUnit case is implemented by arfNode. The BCat case is implemented by recursively applying arfBlock to the two sub-blocks, threading the output fact from the first as the input to the second. Each recursive call produces a rewritten graph; we concatenate them with RGCatO.

Function arfGraph is equally straightforward:

```
arfGraph :: Edges n => ARF (Graph n) n
arfGraph _    GNil        f = return (RGNil, f)
arfGraph pass (GUnit blk) f = arfBlock pass blk f
arfGraph pass (GMany NothingO body NothingO) f
  = do { (body', fb) <- arfBody pass body f
       ; return (body', fb) }
arfGraph pass (GMany NothingO body (JustO exit)) f
  = do { (body', fb) <- arfBody  pass body f
       ; (exit', fx) <- arfBlock pass exit fb
       ; return (body' `RGCatC` exit', fx) }
 --  ... two more equations for GMany ...
```

The pattern is the same as for `arfBlock`: thread facts through the sequence, and concatenate the results. Because the constructors of type `RG` are more polymorphic than those of `Graph`, type `RG` can represent graphs more simply than `Graph`; for example, each element of a `GMany` becomes a single `RG` object, and these `RG` objects are then concatenated to form a single result of type `RG`.

5.3 Analyzing and rewriting nodes

Although interleaving analysis with transformation is tricky, we have succeeded in isolating the algorithm in just two functions, `arfNode` and its backward analog, `arbNode`:

```
arfNode :: Edges n => ARF n n
arfNode pass n f
 = do { mb_g <- withFuel (fp_rewrite pass n f)
      ; case mb_g of
          Nothing -> return (RGUnit f (BUnit n),
                             fp_transfer pass n f)
          Just (FwdRes ag rw) ->
            do { g <- graphOfAGraph ag
               ; let pass' = pass { fp_rewrite = rw }
               ; arfGraph pass' g f } }
```

The code here is more complicated, but still admirably brief. Using the `fp_rewrite` record selector (Figure 3), we begin by extracting the rewriting function from the `FwdPass`, and we apply it to the node `n` and the incoming fact `f`.

The resulting `Maybe` is passed to `withFuel`, which deals with fuel accounting:

```
  withFuel :: Maybe a -> FuelMonad (Maybe a)
```

If `withFuel`'s argument is `Nothing`, *or* if we have run out of optimization fuel (Section 4.7), `withFuel` returns `Nothing`. Otherwise, `withFuel` consumes one unit of fuel and returns its argument (which will be a `Just`). That is all we need say about fuel.

In the `Nothing` case, no rewrite takes place—either because the rewrite function didn't want one or because fuel is exhausted. We return a single-node graph `(RGUnit f (BUnit n))`, decorated with its incoming fact. We also apply the transfer function `(fp_transfer pass)` to the incoming fact to produce the outgoing fact. (Like `fp_rewrite`, `fp_transfer` is a record selector of `FwdPass`.)

In the `Just` case, we receive a replacement `AGraph ag` and a new rewrite function `rw`. We convert `ag` to a `Graph`, using

```
graphOfAGraph :: AGraph n e x -> FuelMonad (Graph n e x)
```

and we analyze the resulting `Graph` with `arfGraph`. This analysis uses `pass'`, which contains the original lattice and transfer function from `pass`, together with the new rewrite function `rg`.

And that's it! If the client wanted deep rewriting, it is implemented by the call to `arfGraph`; if the client wanted shallow rewriting, the rewrite function will have returned `noFwdRw` as `rw`, which is implanted in `pass'` (Section 4.4).

5.4 Fixed points

Lastly, `arfBody` deals with the fixed-point calculation. This part of the implementation is the only really tricky part, and it is cleanly separated from everything else:

```
arfBody  :: Edges n
         => FwdPass n f -> Body n -> FactBase f
         -> FuelMonad (RG n f C C, FactBase f)
arfBody pass body fbase
  = fixpoint (fp_lattice pass) (arfBlock pass) fbase $
    forwardBlockList (factBaseLabels fbase) body
```

Function `forwardBlockList` takes a list of possible entry points and `Body`, and it returns a linear list of blocks, sorted into an order that makes forward dataflow efficient:

```
forwardBlockList
  :: Edges n => [Label]
  -> Body n -> [(Label,Block n C C)]
```

For example, if the `Body` starts at block `L2`, and `L2` branches to `L1`, but not vice versa, then Hoopl will reach a fixed point more quickly if we process `L2` before `L1`. To find an efficient order, `forwardBlockList` uses the methods of the `Edges` class—`entryLabel` and `successors`—to perform a reverse depth-first traversal of the control-flow graph. The order of the blocks does not affect the fixed point or any other part of the answer; it affects only the number of iterations needed to reach the fixed point.

How do we know what entry points to pass to `forwardBlockList`? We treat any block with an entry in the in-flowing `FactBase` as an entry point.

The rest of the work is done by `fixpoint`, which is shared by both forward and backward analyses:

```
fixpoint :: forall n f.
     Edges n
  => Bool     -- going Forward?
  -> DataflowLattice f
  -> (Block n C C -> FactBase f ->
                FuelMonad (RG n f C C, FactBase f))
  -> FactBase f
  -> [(Label, Block n C C)]
  -> FuelMonad (RG n f C C, FactBase f)
```

Except for the mysterious `Bool` passed as the first argument, the type signature tells the story. The third argument is a function that analyzes and rewrites a single block; `fixpoint` applies that function successively to all the blocks, which are passed as the fifth argument. The `fixpoint` function maintains a "Current `FactBase`" which grows monotonically: the initial value of the Current `FactBase` is the fourth argument to `fixpoint`, and the Current `FactBase` is augmented with the new facts that flow out of each `Block` as it is analyzed. The `fixpoint` function keeps analyzing blocks until the Current `FactBase` reaches a fixed point.

The code for `fixpoint` is a massive 70 lines long; for completeness, it appears in Appendix A. The code is mostly straightforward, although we try to be a bit clever about deciding when a new fact means that another iteration over the blocks will be required. There is one more subtle point worth mentioning, which we highlight by considering a forward analysis of this graph, where execution starts at `L1`:

```
  L1: x:=3; goto L4
  L2: x:=4; goto L4
  L4: if x>3 goto L2 else goto L5
```

Block `L2` is unreachable. But if we naïvely process all the blocks (say in order `L1`, `L4`, `L2`), then we will start with the bottom fact for

L2, propagate {x=4} to L4, where it will join with {x=3} to yield {x=⊤}. Given x=⊤, the conditional in L4 cannot be rewritten, and L2 seems reachable. We have lost a good optimization.

Our implementation solves this problem through a clever trick that is safe only for a forward analysis; `fixpoint` analyzes a block only if the block is reachable from an entry point. This trick is not safe for a backward analysis, which is why `fixpoint` takes a `Bool` as its first argument: it must know if the analysis goes forward.

Although the trick can be implemented in just a couple of lines of code, the reasoning behind it is quite subtle—exactly the sort of thing that should be implemented once in Hoopl, so clients don't have to worry about it.

6. Related work

While there is a vast body of literature on dataflow analysis and optimization, relatively little can be found on the *design* of optimizers, which is the topic of this paper. We therefore focus on the foundations of dataflow analysis and on the implementations of some comparable dataflow frameworks.

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).[5] Kam and Ullman (1977) generalize to some monotone functions. Each client of Hoopl must guarantee monotonicity.

Cousot and Cousot (1977, 1979) 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.

Muchnick (1997) presents many examples of both particular analyses and related algorithms.

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

Frameworks Most dataflow frameworks support only analysis, not transformation. The framework computes a fixed point of transfer functions, and it is up to the client of the framework to use that fixed point for transformation. Omitting transformation makes it much easier to build frameworks, and one can find a spectrum of designs. We describe two representative designs, then move on to the prior frameworks that support interleaved analysis and transformation.

The CIL toolkit (Necula et al. 2002) provides an analysis-only framework for C programs. The framework is limited to one representation of control-flow graphs and one representation of instructions, both of which are provided by the framework. The API is complicated; much of the complexity is needed to enable the client to affect which instructions the analysis iterates over.

The Soot framework is designed for analysis of Java programs (Vallée-Rai et al. 2000). While Soot's dataflow library supports only analysis, not transformation, we found much to admire in its design. Soot's library is abstracted over the representation of the control-flow graph and the representation of instructions. Soot's interface for defining lattice and analysis functions is like our own, although because Soot is implemented in an imperative style, additional functions are needed to copy lattice elements. Like CIL, Soot provides only analysis, not transformation.

[5] 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.

The Whirlwind compiler contains the dataflow framework implemented by Lerner, Grove, and Chambers (2002), who were the first to interleave analysis and transformation. Their implementation is much like our early efforts: it is a complicated mix of code that simultaneously manages interleaving, deep rewriting, and fixed-point computation. By separating these tasks, our implementation simplifies the problem dramatically. Whirlwind's implementation also suffers from the difficulty of maintaining pointer invariants in a mutable representation of control-flow graphs, a problem we have discussed elsewhere (Ramsey and Dias 2005).

Because speculative transformation is difficult in an imperative setting, Whirlwind's implementation is split into two phases. The first phase runs the interleaved analyses and transformations to compute the final dataflow facts and a representation of the transformations that should be applied to the input graph. The second phase executes the transformations. In Hoopl, because control-flow graphs are immutable, speculative transformations can be applied immediately, and there is no need for a phase distinction.

In previous work (Ramsey and Dias 2005), we described a zipper-based representation of control-flow graphs, stressing the advantages of immutability. Our new representation, described in Section 3, is a significant improvement:

- We can concatenate nodes, blocks, and graphs in constant time.
- We can do a backward analysis without having to "unzip" (and allocate a copy of) each block.
- Using GADTs, we can represent a flow-graph node using a single type, instead of the triple of first, middle, and last types used in our earlier representation. This change simplifies the interface significantly: instead of providing three transfer functions and three rewrite functions per pass—one for each type of node—a client of Hoopl provides only one transfer function and one rewrite function per pass.
- 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 we tried to compensate for the errors—but the compensation code harbored subtle faults, which we discovered while developing a new back end for the Glasgow Haskell Compiler.

The implementation of Hoopl is also much better than our earlier implementations. Not only is the code simpler conceptually, but it is also shorter: our new implementation is about a third as long as the previous version, which is part of GHC, version 6.12.

7. What we learned

We have spent six years implementing and reimplementing frameworks for dataflow analysis and transformation. This formidable design problem taught us two kinds of lessons: we learned some very specific lessons about representations and algorithms for optimizing compilers, and we were forcibly reminded of some very general, old lessons that are well known not just to functional programmers, but to programmers everywhere.

Our main goal for Hoopl was to combine three good ideas (interleaved analysis and transformation, optimization fuel, and an applicative control-flow graph) in a way that could easily be reused by many, many compiler writers. Reuse requires abstraction, and as is well known, designing good abstractions is challenging. Hoopl's data types and the functions over those types have been through *dozens* of revisions. As we were refining our design, we found it invaluable to operate in two modes: In the first mode, we designed, built, and used a framework as an important component of a real compiler (first Quick C--, then GHC). In the second mode, we designed and built a standalone library, then redesigned and rebuilt it, sometimes going through several significant changes in a week. Operating in the first mode—inside a live compiler—forced us to make sure that no corners were cut, that we were solving a real problem, and that we did not inadvertently cripple some other part of the compiler. Operating in the second mode—as a standalone library—enabled us to iterate furiously, trying out many more ideas than would have been possible in the first mode. We have learned that alternating between these two modes leads to a better design than operating in either mode alone.

We were forcibly reminded of timeless truths: that interfaces are more important than implementations, and that data is more important than code. These truths are reflected in this paper, in which we have given Hoopl's API three times as much space as Hoopl's implementation.

We were also reminded that Haskell's type system (polymorphism, GADTs, higher-order functions, type classes, and so on) is a remarkably effective language for thinking about data and code—and that Haskell lacks a language of interfaces (like ML's signatures) that would make it equally effective for thinking about APIs at a larger scale. Still, as usual, the types were a remarkable aid to writing the code: when we finally agreed on the types presented above, the code almost wrote itself.

Types are widely appreciated at ICFP, but here are three specific examples of how types helped us:

- Reuse is enabled by representation-independence, which in a functional language is expressed through parametric polymorphism. Making Hoopl polymorphic in the nodes 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 nodes, and to embody that knowledge in the `Edges` type class (Section 3.5).
- We are proud of using GADTs to track the open and closed shapes of nodes, blocks, and graphs at compile time. Shapes may seem like a small refinement, but they helped tremendously when building Hoopl, and we expect them to help clients. Giving the *same* shapes to nodes, blocks, and graphs helped our thinking and helped to structure the implementation.
- In our earlier designs, graphs were parameterized over *three* node types: first, middle, and last nodes. Those designs therefore required three transfer functions, three rewrite functions, and so on. Moving to a single, "shapely" node type was a major breakthrough: not only do we have just one node type, but our client need supply only one transfer function and one rewrite function. To make this design work, however, we *must* have the type-level function for `Fact` (Figure 3), to express how incoming and outgoing facts depend on the shape of a node.

Dataflow optimization is usually described as a way to improve imperative programs by mutating control-flow graphs. Such transformations appear very different from the tree rewriting that functional languages are so well known for, and that makes functional languages so attractive for writing other parts of compilers. But even though dataflow optimization looks very different from what we are used to, writing a dataflow optimizer in a pure functional language was a huge win. In a pure functional language, not only do we know that no data structure will be unexpectedly mutated, but we are forced to be explicit about every input and output, and we are encouraged to implement things compositionally. This kind of thinking has helped us make significant improvements to the already tricky work of Lerner, Grove, and Chambers: per-function control of shallow vs deep rewriting (Section 4.4), combinators for dataflow passes (Section 4.5), optimization fuel (Section 4.7), and transparent management of unreachable blocks (Section 5.4). We trust that these improvements are right only because they are implemented in separate parts of the code that cannot interact except through explicit function calls. With this new, improved design in hand, we are now moving back to live-compiler mode, pushing Hoopl into version 6.13 of the Glasgow Haskell Compiler.

Acknowledgments

The first and second authors were funded by a grant from Intel Corporation and by NSF awards CCF-0838899 and CCF-0311482. These authors also thank Microsoft Research Ltd, UK, for funding extended visits to the third author.

References

Andrew W. Appel. 1998. *Modern Compiler Implementation*. Cambridge University Press, Cambridge, UK. Available in three editions: C, Java, and ML.

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.

Patrick Cousot and Radhia Cousot. 1979 (January). Systematic design of program analysis frameworks. In *Conference Record of the 6th Annual ACM Symposium on Principles of Programming Languages*, pages 269–282.

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.

Steven S. Muchnick. 1997. *Advanced compiler design and implementation*. Morgan Kaufmann, San Mateo, CA.

George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. 2002. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228, London, UK. Springer-Verlag.

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.

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.

Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. 2000. Optimizing Java bytecode using the Soot framework: Is it feasible? In *CC '00: Proceedings of the 9th International Conference on Compiler Construction*, pages 18–34, London, UK. Springer-Verlag.

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

A. Code for `fixpoint`

```
data TxFactBase n f
  = TxFB { tfb_fbase :: FactBase f
         , tfb_rg  :: RG n f C C -- Transformed blocks
         , tfb_cha   :: ChangeFlag
         , tfb_lbls  :: LabelSet }
 -- Set the tfb_cha flag iff
 --   (a) the fact in tfb_fbase for or a block L changes
 --   (b) L is in tfb_lbls.
 -- The tfb_lbls are all Labels of the *original*
 -- (not transformed) blocks

updateFact :: DataflowLattice f -> LabelSet -> (Label, f)
           -> (ChangeFlag, FactBase f)
           -> (ChangeFlag, FactBase f)
updateFact lat lbls (lbl, new_fact) (cha, fbase)
  | NoChange <- cha2        = (cha,        fbase)
  | lbl `elemLabelSet` lbls = (SomeChange, new_fbase)
  | otherwise               = (cha,        new_fbase)
  where
    (cha2, res_fact)
      = case lookupFact fbase lbl of
          Nothing -> (SomeChange, new_fact)
          Just old_fact -> fact_extend lat old_fact new_fact
    new_fbase = extendFactBase fbase lbl res_fact

fixpoint :: forall n f. Edges n
         => Bool        -- Going forwards?
         -> DataflowLattice f
         -> (Block n C C -> FactBase f
             -> FuelMonad (RG n f C C, FactBase f))
         -> FactBase f -> [(Label, Block n C C)]
         -> FuelMonad (RG n f C C, FactBase f)
fixpoint is_fwd lat do_block init_fbase blocks
 = do { fuel <- getFuel
      ; tx_fb <- loop fuel init_fbase
      ; return (tfb_rg tx_fb,
                tfb_fbase tx_fb `delFromFactBase` blocks) }
          -- The outgoing FactBase contains facts only for
          -- Labels *not* in the blocks of the graph
 where
  tx_blocks :: [(Label, Block n C C)]
            -> TxFactBase n f -> FuelMonad (TxFactBase n f)
  tx_blocks []             tx_fb = return tx_fb
  tx_blocks ((lbl,blk):bs) tx_fb = tx_block lbl blk tx_fb
                                   >>= tx_blocks bs

  tx_block :: Label -> Block n C C
           -> TxFactBase n f -> FuelMonad (TxFactBase n f)
  tx_block lbl blk tx_fb@(TxFB { tfb_fbase = fbase
                               , tfb_lbls  = lbls
                               , tfb_rg    = blks
                               , tfb_cha   = cha })
    | is_fwd && not (lbl `elemFactBase` fbase)
    = return tx_fb    -- Note [Unreachable blocks]
    | otherwise
    = do { (rg, out_facts) <- do_block blk fbase
         ; let (cha',fbase')
                 = foldr (updateFact lat lbls) (cha,fbase)
                         (factBaseList out_facts)
         ; return (TxFB { tfb_lbls = extendLabelSet lbls lbl
                        , tfb_rg   = rg `RGCatC` blks
                        , tfb_fbase = fbase'
                        , tfb_cha = cha' }) }

  loop :: Fuel -> FactBase f -> FuelMonad (TxFactBase n f)
  loop fuel fbase
    = do { let init_tx_fb = TxFB { tfb_fbase = fbase
                                 , tfb_cha   = NoChange
                                 , tfb_rg    = RGNil
                                 , tfb_lbls  = emptyLabelSet}
         ; tx_fb <- tx_blocks blocks init_tx_fb
         ; case tfb_cha tx_fb of
             NoChange   -> return tx_fb
             SomeChange -> setFuel fuel >>
                           loop fuel (tfb_fbase tx_fb) }
```

B. 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 D.

`Add :: Operator` not shown (but see page 7).
`AGraph` defined in Figure 5 on page 6.
`analyzeAndRewriteFwd` defined on page 9.
`arbNode` defined on page 10.
`ARF` defined on page 9.
`arfBlock` defined on page 9.
`arfBody` defined on page 10.
`arfGraph` defined on page 10.
`arfNode` defined on page 10.
`Assign` defined in Figure 1 on page 3.
`B` defined in Figure 6 on page 7.
`b1` let- or λ-bound on page 4.
`b2` let- or λ-bound on page 4.
`BCat` defined in Figure 2 on page 3.
`Binop :: Operator -> Expr -> Expr -> Expr` not shown (but see page 7).
`Block` defined in Figure 2 on page 3.
`Body` defined in Figure 2 on page 3.
`BodyCat` defined in Figure 2 on page 3.
`BodyEmpty` defined in Figure 2 on page 3.
`BodyUnit` defined in Figure 2 on page 3.
`Branch` defined in Figure 1 on page 3.
`BUnit` defined in Figure 2 on page 3.
`C` defined in Figure 2 on page 3.
`ChangeFlag` defined in Figure 3 on page 4.
`CondBranch` defined in Figure 1 on page 3.
`ConstFact` defined in Figure 6 on page 7.
`constFactAdd` defined in Figure 6 on page 7.
`constLattice` defined in Figure 6 on page 7.
`constProp` defined in Figure 6 on page 7.
`constPropPass` defined in Figure 6 on page 7.
`DataflowLattice` defined in Figure 3 on page 4.
`delFromFactBase :: FactBase f -> [(Label,a)] -> FactBase f` not shown (but see page 13).
`Edges` defined in Figure 2 on page 3.
`elemFactBase :: Label -> FactBase f -> Bool` not shown (but see page 13).
`elemLabelSet :: Label -> LabelSet -> Bool` not shown (but see page 13).
`emptyLabelSet :: LabelSet` not shown (but see page 13).
`entryLabel` defined in Figure 2 on page 3.
`extendFactBase :: FactBase f -> Label -> f -> FactBase f` not shown (but see page 13).
`extendLabelSet :: LabelSet -> Label -> LabelSet` not shown (but see page 13).
`Fact` defined in Figure 3 on page 4.
`FactBase` defined in Figure 3 on page 4.
`factBaseLabels :: FactBase f -> [Label]` not shown (but see page 13).
`fact_bot` defined in Figure 3 on page 4.
`fact_extend` defined in Figure 3 on page 4.
`fixpoint` defined on page 10.
`forwardBlockList` defined on page 10.
`fp_lattice` defined in Figure 3 on page 4.

`fp_rewrite` defined in Figure 3 on page 4.
`fp_transfer` defined in Figure 3 on page 4.
`Fuel` defined on page 10.
`FuelMonad` defined on page 8.
`FwdPass` defined in Figure 3 on page 4.
`FwdRes` defined in Figure 3 on page 4.
`FwdRewrite` defined in Figure 3 on page 4.
`FwdTransfer` defined in Figure 3 on page 4.
`gCat` defined on page 4.
`getFuel :: FuelMonad Fuel` not shown (but see page 13).
`GMany` defined in Figure 2 on page 3.
`GNil` defined in Figure 2 on page 3.
`Graph` defined in Figure 2 on page 3.
`graphOfAGraph` defined on page 10.
`GUnit` defined in Figure 2 on page 3.
`gUnitCO` defined in Figure 5 on page 6.
`gUnitOC` defined in Figure 5 on page 6.
`HasConst` defined in Figure 6 on page 7.
`I` defined in Figure 6 on page 7.
`init_fbase` let- or λ-bound on page 13.
`JustO` defined in Figure 2 on page 3.
`Label` defined in Figure 2 on page 3.
`LabelMap` (a type) not shown (but see page 13).
`LabelNode` defined in Figure 1 on page 3.
`LabelSet` (a type) not shown (but see page 13).
`lookupFact :: FactBase f -> Label -> Maybe f` not shown (but see page 13).
`loop` defined on page 13.
`mapE` defined in Figure 6 on page 7.
`MaybeO` defined in Figure 2 on page 3.
`mkBranch` defined in Figure 5 on page 6.
`mkFactBase :: [(Label, f)] -> FactBase f` not shown (but see page 4).
`mkIfThenElse` defined in Figure 5 on page 6.
`mkLabel` defined in Figure 5 on page 6.
`new_fbase` let- or λ-bound on page 13.
`NoChange` defined in Figure 3 on page 4.
`Node` defined in Figure 1 on page 3.
`noFwdRw` defined on page 7.
`normalizeBody` defined on page 9.
`NothingO` defined in Figure 2 on page 3.
`O` defined in Figure 2 on page 3.
`res_fact` let- or λ-bound on page 13.
`rewriteE` defined in Figure 6 on page 7.
`RG` defined in Figure 7 on page 9.
`RGCatC` defined in Figure 7 on page 9.
`RGCatO` defined in Figure 7 on page 9.
`RGNil` defined in Figure 7 on page 9.
`RGUnit` defined in Figure 7 on page 9.
`simplify` defined in Figure 6 on page 7.
`SomeChange` defined in Figure 3 on page 4.
`Store` defined in Figure 1 on page 3.
`successors` defined in Figure 2 on page 3.
`tfb_cha` defined on page 13.
`tfb_fbase` defined on page 13.
`tfb_lbls` defined on page 13.
`tfb_rg` defined on page 13.
`thenFwd` defined on page 7.
`thenFwdRw` defined on page 7.
`Top` defined in Figure 6 on page 7.
`tx_block` defined on page 13.
`tx_blocks` defined on page 13.
`TxFactBase` defined on page 13.
`TxFB` defined on page 13.
`updateFact` defined on page 13.
`varHasConst` defined in Figure 6 on page 7.
`withFuel` defined on page 10.
`withLabels` defined in Figure 5 on page 6.

C. Undefined identifiers

`ag` (p10), `blk` (p10), `blks` (p13), `blocks` (p13), `body` (p9), `body'` (p10), `bs` (p4), `bs1` (p4), `bs2` (p4), `cha` (p13), `cha'` (p13), `cha2` (p13), `do_block` (p13), `ex` (p3), `exit` (p10), `exit'` (p10), `Expr` (p3), `factBaseList` (p13), `facts` (p7), `family` (p4), `fb` (p10), `fbase` (p10), `fbase'` (p13), `fid` (p7), `fmap` (p7), `fn` (p6), `fuel` (p13), `fx` (p10), `goto` (p2), `init_tx_fb` (p13), `is_fwd` (p13), `iterFwdRw` (p7), `lat` (p13), `lbl` (p13), `lbls` (p13), `ls` (p6), `M.lookup` (p7), `Map.empty` (p7), `Map.insert` (p7), `Map.Map` (p7), `mb_g` (p10), `new` (p7), `new_fact` (p13), `node` (p6), `old` (p7), `old_fact` (p13), `out_facts` (p13), `pass` (p9), `pass'` (p10), `rg` (p9), `rw` (p7), `rw1` (p7), `rw1a` (p7), `rw2` (p7), `setFuel` (p13), `s_exp` (p7), `stdMapJoin` (p7), `thing` (p9), `tid` (p7), `toAGraph` (p7), `transfer` (p8), `transfer_fn` (p6), `tx_fb` (p13), `Var` (p3).

D. Identifiers defined in Haskell Prelude

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