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

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 entangled with the details of a particular compiler. We describe Hoopl, a reusable library that makes it unusually easy to define new analyses and transformations for any compiler written in Haskell. Hoopl’s interface is modular and polymorphic, and it offers unusually strong static guarantees. The implementation 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Categories and Subject Descriptors D.3.4 [Processors]: Optimization, Compilers; D.3.2 [Language Classifications]: Applicative (functional) languages, Haskell

General Terms Algorithms, Design, Languages

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 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 is 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. Although such libraries exist, as we discuss in Section 5, they have complex APIs and implementations, and none interleave analysis with transformation.

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. Although pure functional languages are not obviously suited to writing standard algorithms that transform control-flow graphs, pure functional code is actually easier to write, and far easier to write correctly, than code that is mostly functional but uses a mutable representation of graphs [Ramsey and Dias 2008]. When analysis and transformation are interleaved, so that graphs must be transformed speculatively, without knowing whether a transformed graph will be retained or discarded, pure functional code offers even more benefits.

- 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, simple API, which hides a subtle implementation (Sections 3 and 4). You provide a representation for facts, a transfer function that transforms facts across nodes, and a rewrite function that can use a fact to justify rewriting a node. Hoopl “lifts” these node-level functions to work over control-flow graphs, solves recursion equations, and interleaves rewriting with analysis. Designing APIs is surprisingly hard; after a dozen significantly different iterations, we offer our API as a contribution.

- Because clients can perform very local reasoning (“y is live before x:=y+2”), analyses and transformations built on Hoopl are small, simple, and easy to get right. Moreover, Hoopl helps you write correct optimizations: statically, it rules out transformations that violate invariants of the control-flow graph (Sections 3 and 4), and dynamically, it can help find the first transformation that introduces a fault in a test program (Section 5).

- Hoopl implements subtle algorithms, including (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. In this paper, each tricky piece is handled in just one place, separate from the others (Section 5). We emphasize this implementation as an object of interest in its own right.

Our work bridges the gap between abstract, theoretical presentations and actual compilers. Hoopl is available from http://ghc.cs.tufts.edu/hoopl and also from Hackage (version 3.8.6.0). One of Hoopl’s clients is the Glasgow Haskell Compiler, which uses Hoopl to optimize imperative code in GHC’s back end.

Hoopl’s API is made possible by sophisticated aspects of Haskell’s type system, such as higher-rank polymorphism, GADTs, and type functions. Hoopl may therefore also serve as a case study in the utility of these features.
2. Dataflow analysis & transformation by example

A control-flow graph, perhaps 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 dataflow facts, then use those facts to justify code-improving transformations (or rewrites) on a control-flow graph.

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

Before Facts After

```
\begin{tabular}{ll}
\hline
x := 3+4 & x := 7 \\
\hline
z := x>5 & z := True \\
\hline
if z then goto L1 \\
else goto L2
\end{tabular}
```

Constant propagation works from top to bottom. In this example, we start with the empty fact. Given that fact and the node \( x:=3+4 \), can we make a transformation? Yes: constant folding 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 make another transformation? Yes: constant folding can replace the node with \( z:=True \). 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 it has only straight-line code; control flow makes dataflow analysis more complicated. For example, consider a graph with a conditional statement, starting at L1:

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 (L1 and L2), we must join the facts coming from those two places. All paths to L3 produce the fact \( y=4 \), so we can conclude that \( y=4 \) at L3. But depending on the path to L3, \( x \) may have different values, so we conclude \( \{x=T\} \), meaning that there is no single value held by \( x \) at L3. The final result of joining the dataflow facts that flow to L3 is the fact \( \{x=T \land y=4 \land z=T\} \).

Forward and backward. Constant propagation works forward, and a fact is often an assertion about the program state, such as "variable \( x \) holds value \( 7 \)." Some useful analyses work backward. 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.

Interleaved analysis and transformation. Our first example interleaves analysis and transformation. Interleaving makes it easy to write effective analyses. If instead we had to finish analyzing the block before transforming it, analyses would have to "predict" the results of 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 \( z:=True \), then apply the transfer function to the new node, the transfer function becomes wonderfully simple: it merely has to see if the right hand side is a constant. You can see code in Section 4.6.

Another example is the interleaving of liveness analysis and dead-code elimination. As mentioned in Section 1 it is sufficient for the analysis to say "\( y \) is live before \( x:=y+2 \). 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 transformed away (eliminated). When several analyses and transformations can interact, interleaving them offers even more compelling benefits; for more substantial examples, consult [Lerner, Grove, and Chambers 2002].

But the benefits come at a cost. To compute valid facts for a program that has loops, an analysis may require multiple iterations. Before the final iteration, the analysis may compute a fact that is invalid, and a transformation may use the invalid fact to rewrite the program (Section 4.7). To avoid unjustified rewrites, any rewrite based on an invalid fact must be rolled back; transformations must be speculative. As described in Section 4.7 Hoopl manages speculation with minimal cooperation from the client.

While it is wonderful that we can create complex optimizations by interleaving very simple analyses and transformations, it is not so wonderful that very simple analyses and transformations, when interleaved, can exhibit complex emergent behavior. Because such behavior is not easily predicted, it is essential to have good tools for debugging. Hoopl's primary debugging tool is an implementation of Whalley's search technique for finding fault-inducing transformations (Section 5.3).

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 3.3).
- A basic block is a sequence of nodes (Section 3.3).
- A graph is an arbitrarily complicated control-flow graph: basic blocks connected by edges (Section 3.4).

3.1 Shapes: Open and closed

In Hoopl, nodes, blocks, and graphs share an important new property: a shape. A thing's shape tells us whether the thing is open or closed on entry and open or closed on exit. At an open point, control may implicitly "fall through;" at a closed point, control transfer must be explicit and to a named label. 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.
- The shape of a function-call node is up to the client. If a call always returns to its inline successor, it could be open on
data Node e x where
    Label :: Label -> Node C O
    Assign :: Var -> Expr -> Node O O
    Store :: Expr -> Expr -> Node O O
    Branch :: Label -> Node O C
    Cond :: Expr -> Label -> Label -> Node O C
... more constructors ...

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

entry and exit. But if a call could return in multiple ways—for example by returning normally or by raising an exception—then it has to be closed on exit. GHC uses calls of both shapes.

Blocks and graphs have shapes too. For example the block

\[
x := 7;\ y := x + 2;\ \text{goto}\ L
\]

is open on entry and closed on exit, which we often abbreviate “open/closed.” We may also 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 on entry, it has a unique predecessor; if it is closed, it may have arbitrarily many predecessors—or none.
- If E is open on 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 control-flow graph are nodes. For example, in a back end a node might represent a machine instruction, such as a load, a call, or a conditional branch; in a higher-level intermediate form, a node might represent a simple statement. 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 data specified by the client, including (say) method calls, C statements, stack maps, or whatever.

The type of a node specifies its shape \textit{at compile time}. Concretely, the type constructor for a node has kind \texttt{\(\text{Node}\ \text{e}\ \text{O}\ \text{C}\)}, where the two type parameters are type-level flags, one for entry and one for exit. Each type parameter may be instantiated only with type \texttt{O} (for open) or type \texttt{C} (for closed).

As an example, Figure 1 shows a typical node type as it might be defined 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 \texttt{Label} takes a \texttt{Label} and returns a node of type \texttt{Node C O}, where the “\(C\)” says “closed on entry” and the “\(O\)” says “open on exit”. The types \texttt{Label}, \texttt{O}, and \texttt{C} are defined by Hoopl (Figure 2). In other examples from Figure 1, constructor \texttt{Assign} takes a variable and an expression, and it returns a node open on both entry and exit; constructor \texttt{Store} is similar. Finally, control-transfer nodes \texttt{Branch} and \texttt{Cond} (conditional branch) are open on entry and closed on exit. Types \texttt{Var} and \texttt{Expr} are private to the client, and Hoopl knows nothing about them.

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. Because of the position each shape of node occupies in a basic block, we often call them \texttt{first}, \texttt{middle}, and \texttt{last} nodes respectively.

1 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

data Block n e x where
    BFirst :: n C O -> Block n C O
    BMiddle :: n O O -> Block n O O
    BLast :: n C O -> Block n C O
    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)
        -> LabelMap (Block n C C)
        -> MaybeO x (Block n C O)
        -> Graph n e x

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

eventype Label -- abstract
neventype LabelMap a -- finite map from Label to a
addBlock :: NonLocal n
        => Block n C C
        -> LabelMap (Block n C C)
        -> LabelMap (Block n C C)
blockUnion :: LabelMap a -> LabelMap a -> LabelMap a

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

Figure 2. The block and graph types defined by Hoopl

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 flag types that make it open or closed at entry and exit.

The \texttt{BFirst}, \texttt{BMiddle}, and \texttt{BLast} constructors create one-node blocks. Each of these constructors is polymorphic in the node’s \texttt{representation} but monomorphic in its \texttt{shape}. Why not use a single constructor of type \(n e x \rightarrow \text{Block n e x}\), which would be polymorphic in a node’s representation and shape? Because by making the shape known statically, we simplify the implementation of analysis and transformation in Section 5.

The \texttt{BCat} constructor 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 \texttt{BCat}, whose first argument must be open on exit and whose second argument must be open on entry. It is impossible, for example, to concatenate a \texttt{Branch} immediately before an \texttt{Assign}. Indeed, the \texttt{Block} type guarantees statically that any closed/closed \texttt{Block}—which compiler writers normally call a “basic block”—consists of exactly one first node (such as \texttt{Label} in Figure 1), followed by zero or more middle nodes (\texttt{Assign} or \texttt{Store}), and terminated with exactly one last node (\texttt{Branch} or \texttt{Cond}). Enforcing these invariants by using GADTs is one of Hoopl’s innovations.

block. This representation is standard \cite{appel98}, and it costs nothing in practice: such code is easily sequentialized without superfluous branches.
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 over its shape at entry and exit (e and x). Graph 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 GU nit. 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 on 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 on entry. We express our compile-time knowledge by using the type Maybe0 e (Block n O C), a type-indexed version of Maybe which is also defined in Figure 2; the type Maybe0 O a is isomorphic to a, while the type Maybe0 C a is isomorphic to ().

- The body of the graph is a collection of closed/closed blocks. To facilitate traversal of the graph, we represent the body as a finite map from label to block.

- 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 can be spliced together nicely; the cost is logarithmic in the number of closed/closed blocks. Unlike blocks, two graphs may be spliced together not only when they are both open at splice point but also when they are both closed—and not in the other two cases:

\[
gSplice :: \text{Graph} \ n \ e \ a \rightarrow \text{Graph} \ n \ a \ x \rightarrow \text{Graph} \ n \ e \ x
\]

\[
gSplice \ GNil g2 = g2
\]

\[
gSplice \ (GU nit \ b1) \ (GU nit \ b2) = GU nit \ (b1 \ 'BCat' \ b2)
\]

\[
gSplice \ (GU nit \ b) \ (GMany (Just0 \ e) \ bs \ x) = GMany \ (Just0 \ (b \ 'BCat' \ e)) \ bs \ x
\]

\[
gSplice \ (GMany \ e \ bs \ (Just0 \ x)) \ (GU nit \ b2) = GMany \ e \ bs \ (Just0 \ (x \ 'BCat' \ b2))
\]

\[
gSplice \ (GMany \ e1 \ bs1 \ (Just0 \ x1)) \ (GMany \ (Just0 \ e2) \ bs2 \ x2) = GMany \ e1 \ bs1 \ 'blockUnion' \ (b \ 'addBlock' \ bs2) \ x2
\]

\[
\text{where } b = x1 \ 'BCat' \ e2
\]

\[
gSplice \ (GMany \ e1 \ bs1 \ NothingO) \ (GMany \ NothingO \ bs2 \ x2) = GMany \ e1 \ bs1 \ 'blockUnion' \ bs2 \ x2
\]

This definition illustrates the power of GADTs: the pattern matching is exhaustive, and all the shape invariants are checked statically. For example, consider the second-to-last equation for gSplice. Since the exit sequence of the first argument is Just0 x1, we know that type parameter a is 0, and hence the entry sequence of the second argument must be Just0 e2. Moreover, block e1 must be closed/open, and block e2 must be open/closed. We can therefore concatenate x1 and e2 with BCat to produce a closed/closed block b, which is added to the body of the result.

We have carefully crafted the types so that if BCat is considered as an associative operator, every graph has a unique representation. To guarantee uniqueness, GU nit is restricted to open/open blocks. If GU nit were more polymorphic, 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.

<table>
<thead>
<tr>
<th>Part of optimizer</th>
<th>Specified by</th>
<th>Implemented by</th>
<th>How many</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control-flow graphs</td>
<td>US</td>
<td>US</td>
<td>One</td>
</tr>
<tr>
<td>Nodes in a control-flow graph</td>
<td>YOU</td>
<td>YOU</td>
<td>One type per language</td>
</tr>
<tr>
<td>Dataflow fact F</td>
<td>YOU</td>
<td>YOU</td>
<td>One type per logic</td>
</tr>
<tr>
<td>Lattice operations</td>
<td>US</td>
<td>YOU</td>
<td>One set per logic</td>
</tr>
<tr>
<td>Transfer functions</td>
<td>US</td>
<td>YOU</td>
<td>One per analysis</td>
</tr>
<tr>
<td>Rewrite functions</td>
<td>US</td>
<td>YOU</td>
<td>One per transformation</td>
</tr>
<tr>
<td>Analyze-and-rewrite functions</td>
<td>US</td>
<td>US</td>
<td>Two (forward, backward) functions</td>
</tr>
</tbody>
</table>

Table 3. Parts of an optimizer built with Hoopl

3.5 Edges, labels and successors

Although Hoopl is polymorphic in the type of nodes, it still needs to know how control may be transferred from one node to another. Within a block, a control-flow edge is implicit in every application of the BCat constructor. An implicit edge originates in a first node or a middle node and flows to a middle node or a last node.

Between blocks, a control-flow edge is represented as chosen by the client. An explicit edge originates in a last node and flows to a (labelled) first node. If Hoopl is polymorphic in the node type, how can it follow such edges? Hoopl requires the client to make the node type an instance of Hoopl’s NonLocal type class, which is defined in Figure 2. The entryLabel method takes a first node (one closed on entry, as per Section 4.2) and returns its Label; the successors method takes a last node (closed on exit) and returns the Labels to which it can transfer control.

In Figure 1, the client’s instance declaration for Node would be

\[
\text{instance NonLocal Node where}
\]

\[
\text{entryLabel (Label l) = l}
\]

\[
\text{successors (Branch b) = [b]}
\]

\[
\text{successors (Cond e b1 b2) = [b1, b2]}
\]

Again, the pattern matching for both functions is exhaustive, and the compiler checks this fact statically. 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 the client provides this information about nodes, it is convenient for Hoopl to get the same information about blocks. Internally, Hoopl uses this instance declaration for the Block type:

\[
\text{instance NonLocal n => NonLocal (Block n) where}
\]

\[
\text{entryLabel (BF irst n) = entryLabel n}
\]

\[
\text{entryLabel (BCat b _) = entryLabel b}
\]

\[
\text{successors (BLast n) = successors n}
\]

\[
\text{successors (BCat _ b) = successors b}
\]

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 NonLocal, because even if a Graph is closed on entry, it need 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; a client must supply these 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 3.1). Each analysis uses facts that are specific to that par-
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, performs a monadic action, and returns either Nothing or Just g, where g is a graph that should replace the node (Sections 4.3 and 4.4).

For many code-improving transformations, the ability to rewrite a node by a graph is crucial.

These requirements are summarized in Table 3. Because facts, transfer functions, and rewrite functions work together, we combine them in a single record of type FwdPass (Figure 4).

Given a node type n and a FwdPass, a client can Hoopl to analyze and rewrite a graph. Hoopl provides a fully polymorphic interface, but for purposes of exposition, we present a function that is specialized to a closed/closed graph:

```haskell
data FwdPass m n f
  = FwdPass fp_lattice :: DataflowLattice f,
    fp_transfer :: FwdTransfer n f,
    fp_rewrite :: FwdRewrite m n f
```

--- Lattice ---

data DataflowLattice f = DataflowLattice
  fact_bot :: f
  , fact_join :: JoinFun f
type JoinFun f = (forall e x . n e x -> f -> Fact x f) -> FwdTransfer n f

--- Transfers ---

newtype FwdTransfer n f -- abstract type
  :: (forall e x . n e x -> f -> Fact x f) -> FwdTransfer n f

--- Rewrites ---

newtype FwdRewrite m n f -- abstract type
  :: (forall e x . n e x -> f -> Fact x f) -> FwdRewrite m n f

--- Fact-like things, aka "fact(s)" ---

type family Fact x f :: *

type instance Fact C f = FactBase f
type instance Fact O f = f

--- FactBase ---

type FactBase f = LabelMap f

--- Optimization fuel ---

type Fuel = Int

class Monad m => FuelMonad m
  where
type instance FuelMonad m

--- Rolling back speculative rewrites ---

class Monad m => CkpointMonad m
  where
type instance CkpointMonad m

--- Checkpoint Monad ---

type instance CkpointMonad m

--- Fuel Monad ---

type instance FuelMonad m

Figure 4. Hoopl API data types

### 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, but in general, dataflow analysis establishes properties of paths:

- An assertion about all paths to a program point is established by a forward 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 backward 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 \( f_{\text{old}} \) is the fact currently associated with a label L, and if a transfer function propagates a new fact \( f_{\text{new}} \) into label L, Hoopl replaces \( f_{\text{old}} \) with the join \( f_{\text{old}} \sqcup f_{\text{new}} \). And Hoopl needs to know if \( f_{\text{old}} \sqcup f_{\text{new}} = f_{\text{old}} \), 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 \( \text{DataflowLattice} f \) (Figure 4). As noted in the previous paragraph, Hoopl needs to know when the result of a join is equal to the old fact. It is often easiest to answer this question while the join itself is being computed. By contrast, a post facto equality test on facts might cost almost as much as a join. For these reasons, Hoopl does not require a separate equality test on facts. Instead, Hoopl requires that \( \text{fact} \_\text{join} \) return a \( \text{ChangeFlag} \) as well as the join. If the join is the same as the old fact, the \( \text{ChangeFlag} \) should be NoChange; if not, the \( \text{ChangeFlag} \) should be SomeChange.

To help clients create lattices and join functions, Hoopl includes functions and constructors that can extend a fact type with top and bottom elements. In this paper, we use only type \( \text{WithTop} \), which comes with value constructors that have these types:

\[
\begin{align*}
\text{PElem} & : f \rightarrow \text{WithTop} f \\
\text{Top} & : \text{WithTop} f
\end{align*}
\]

Hoopl provides combinators which make it easy to create joint functions that use \( \text{Top} \). The most useful is \( \text{extendJoinDomain} \), which uses auxiliary types defined in Figure 4.

\[
\text{extendJoinDomain} :: (\text{OldFact} f \rightarrow \text{NewFact} f \rightarrow (\text{ChangeFlag}, \text{WithTop} f)) \rightarrow \text{JoinFun} (\text{WithTop} f)
\]

A client supplies a join function that consumes only facts of type \( f \), but may produce either \( \text{Top} \) or a fact of type \( f \)—as in the example of Figure 5 below. Calling \( \text{extendJoinDomain} \) extends the client’s function to a proper join function on the type \( \text{WithTop} a \), guaranteeing that joins involving \( \text{Top} \) obey the appropriate algebraic laws.

Hoopl also provides a value constructor \( \text{Bot} \) and type constructors \( \text{WithBot} \) and \( \text{WithTopAndBot} \), along with similar functions. Constructors \( \text{Top} \) and \( \text{Bot} \) are polymorphic, so for example, \( \text{Top} \) also has type \( \text{WithTopAndBot} a \).

It is also common to use a lattice that takes the form of a finite map. In such lattices it is typical to join maps pointwise, and Hoopl provides a function that makes it convenient to do so:

\[
\text{joinMaps} :: \text{Ord} k \rightarrow \text{JoinFun} f \rightarrow \text{JoinFun} (\text{Map.Map} k f)
\]

4.2 The transfer function

A forward transfer function is presented with the dataflow fact coming into a node, and it computes dataflow fact(s) on the node’s outgoing edge(s). In a forward analysis, Hoopl 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 doing constant propagation (Section 4) on the following graph, whose entry point is L1:

\[
\begin{align*}
\text{L1: } & x=3; \text{goto L2} \\
\text{L2: } & y=x+4; x=x-1; \\
& \text{if } x>0 \text{ then goto L2 else return}
\end{align*}
\]

Forward analysis starts with the bottom fact \( \{ \} \) at every label except the entry L1. The initial fact at L1 is \( \{ x=3, y=7 \} \). Analyzing L1 propagates this fact forward, applying the transfer function successively to the nodes of L1, and propagating the new fact \( \{ x=3, y=7 \} \) to L2. This new fact is joined with the existing (bottom) fact at L2. Now the analysis propagates L2’s fact forward, again applying the transfer function, and propagating the new fact \( \{ x=2, y=7 \} \) to L2. Again the new fact is joined with the existing fact at L2, and the process repeats until the facts reach a fixed point.

A transfer function has an unusual sort of type: not quite a dependent type, but not a bog-standard polymorphic type either. The result type of the transfer function is indexed by the shape (i.e., the type) of the node argument: If the node is open on exit, the transfer function produces a single fact. But if the node is closed on exit, the transfer function produces a collection of \( (\text{Label}, \text{fact}) \) pairs: one for each outgoing edge. The collection is represented by a \( \text{FactBase} \); auxiliary function \( \text{mkFactBase} \) (Figure 4) joins facts on distinct outgoing edges that target the same label.

The indexing is expressed by Haskell’s (recently added) indexed type families. A forward transfer function supplied by a client, which is passed to \( \text{mkTransfer} \), is polymorphic in \( e \) and \( x \) (Figure 4). It takes a node of type \( \text{Node} e x \), and it returns a \( \text{fact transformer} \) of type \( f \rightarrow \text{Fact} x f \). Type constructor \( \text{Fact} \) is a species of type-level function: its signature is given in the type family declaration, and its definition is given by two type instance declarations. The first declaration says that a \( \text{Fact} \ f \), which comes out of a node \( \text{open} \) on exit, is just a \( \text{Fact} f \). The second declaration says that a \( \text{Fact C f} \), which comes out of a node \( \text{closed} \) on exit, is a mapping from \( \text{Label} \) to facts.

4.3 The rewrite function and the client’s monad

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, facts about liveness may allow us to replace a dead assignment with a no-op.

A \( \text{FwdPass} \) therefore includes a rewrite function, whose type, \( \text{FwdRewrite} \), is abstract (Figure 4). A programmer creating a rewrite function chooses the type of a node \( n \) and a dataflow fact \( f \). A rewrite function might also want to consume fresh names (e.g., to label new blocks) or take other actions (e.g., logging rewrites). So that a rewrite function may take actions, Hoopl requires that a programmer creating a rewrite function also choose a monad \( m \). So that Hoopl may roll back actions taken by speculative rewrites, the monad must satisfy the constraint \( \text{CkpointMonad} m \), as explained in Section 4.7 below. The programmer may write code that works with any such monad, may create a monad just for the client, or may use a monad supplied by Hoopl.

When these choices are made, the easy way to create a rewrite function is to call the function \( \text{mkF Rewrite} \) in Figure 5. The client supplies a function \( r \), which is specialized to a particular node, fact, and monad, but is polymorphic in the shape of the node to be rewritten. Function \( r \) takes a node and a fact and returns a monadic computation, but what result should that computation return? Returning a new node is not good enough: in general, it must be possible for rewriting to result in a graph. For example,
we might want to remove a node by returning 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 returning a graph containing new blocks with internal control flow.

It must also be possible for a rewrite function to decide to do nothing. The result of the monadic computation returned by \( r \) may therefore be \( \text{Nothing} \), indicating that the node should not be rewritten, or \( \text{Just} \ g \), indicating that the node should be replaced with \( g \); the replacement graph.

The type of \( \text{mkFRewrite} \) in Figure 4 guarantees that the replacement graph \( g \) has the same shape as the node being rewritten. For example, a branch instruction can be replaced only by a graph closed on exit.

4.4 Shallow rewriting, deep rewriting, rewriting combinators, and the meaning of \( \text{FwdRewrite} \)

When a node is rewritten, the replacement graph \( g \) must itself be analyzed, and its nodes may be further rewritten. Hoopl can make a recursive call to \( \text{analyzeAndRewriteFwdBody} \)—but how should it rewrite the replacement graph \( g \)? There are two common cases:

- Rewrite \( g \) using the same rewrite function that produced \( g \). 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 4.5).
- Analyze \( g \) without rewriting it. This procedure is called shallow rewriting.

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 backward 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, through the use of the four combinators \( \text{mkFRewrite}, \text{thenFwdRw}, \text{iterFwdRw}, \) and \( \text{noFwdRw} \) shown in Figure 4. Every rewrite function is made with these combinators, and its behavior is characterized by the answers to two questions: Does the function rewrite a node to a replacement graph? If so, what rewrite function should be used to analyze the replacement graph recursively? To answer these questions, we present an algebraic datatype that models \( \text{FwdRewrite} \) with one constructor for each combinator:

\[
data \text{Rw} \ r = \text{Mk} r | \text{Then} (\text{Rw} \ r) | \text{Iter} (\text{Rw} \ r) | \text{No}
\]

Using this model, we specify how a rewrite function works by giving a reference implementation: the function \( \text{rewrite} \), below, computes the replacement graph and rewrite function that result from applying a rewrite function \( r \) to a node and a fact \( f \). The code is in continuation-passing style; when the node is rewritten, the first continuation \( j \) accepts a pair containing the replacement graph and the new rewrite function to be used to transform it. When the node is not rewritten, the second continuation \( n \) is the (lazily evaluated) result.

\[
\text{rewrite} :: \text{Monad} \ m => \text{FwdRewrite} \ m \ n \ f \to \text{Fact} \ n \ f \to \text{Fact} \ n \ f
\]

\[
\text{rewrite} \ n \ f \ = \ \text{rew} \ r \ \text{return} \ \text{Just} \ \text{return} \ \text{Nothing}
\]

where

\[
\text{rew} \ (\text{Mk} \ r) \ j \ n = \text{do} \ (mg \leftarrow \text{rew} \ n \ f) ; \text{case} \ mg \ \text{of} \ \text{Nothing} \rightarrow \ n ; \text{Just} \ g \rightarrow j \ (g, \text{No})
\]

\[
\text{rew} \ (\text{Then} \ f) \ j \ n = \text{rew} \ r \ j \ (j, \text{add r2}) \ (\text{rew} \ r2 \ j \ n)
\]

\[
\text{rew} \ (\text{Iter} \ r) \ j \ n = \text{rew} \ r \ j \ (j, \text{add (Iter r) n})
\]

\[
\text{rew} \ \text{No} \ j \ n = n
\]

\[
\text{add} \ \text{nextrw} \ (g, r) = (g, r \ '\text{Then}' \ \text{nextrw})
\]

Appealing to this model, we see that

- A function \( \text{mkFRewrite} \) never rewrites a replacement graph; this behavior is shallow rewriting.
- When a function \( r1 \ '\text{thenFwdRw}' \ r2 \) is applied to a node, if \( r1 \) replaces the node, then \( r2 \) is used to transform the replacement graph. And if \( r1 \) does not replace the node, Hoopl iterates \( r2 \).
- When a function \( \text{iterFwdRw} \ r \) rewrites a node, \( \text{iterFwdRw} \ r \) is used to transform the replacement graph; this behavior is deep rewriting. If \( r \) does not rewrite a node, neither does \( \text{iterFwdRw} \ r \).
- Finally, \( \text{noFwdRw} \) never replaces a graph.

For convenience, we also provide the function \( \text{deepFwdRw} \), which is the composition of \( \text{iterFwdRw} \) and \( \text{mkFRewrite} \).

Our combinators satisfy the algebraic laws that you would expect; for example, \( \text{noFwdRw} \) is a left and right identity of \( \text{thenFwdRw} \). A more interesting law is

\[
\text{iterFwdRw} \ r = r \ '\text{thenFwdRw}' \ \text{iterFwdRw} \ r
\]

Unfortunately, this law cannot be used to define \( \text{iterFwdRw} \) if we used this law to define \( \text{thenFwdRw} \), then when \( r \) returned \( \text{Nothing} \), \( \text{iterFwdRw} \ r \) would diverge.

4.5 When the type of nodes is not known

We note above (Section 4.2) that the type of a transfer function’s result depends on the argument’s shape on exit. It is easy for a client to write a type-indexed transfer function, because the client defines the constructor and shape for each node. The client’s transfer functions discriminate on the constructor and so can return a result that is indexed by each node’s shape.

What if you want to write a transfer function that does not know the type of the node? For example, a dominator analysis need not scrutinize nodes; it needs to know only about labels and edges in the graph. Ideally, a dominator analysis would work with any type of node \( n \), provided only that \( n \) is an instance of the \( \text{NonLocal} \) type class.

But if we don’t know the type of \( n \), we can’t write a function of type \( n \to f \to \text{Fact} \ f \), because the only way to get the result type right is to scrutinize the constructors of \( n \).

There is another way; in place of a single function that is polymorphic in shape, Hoopl also accepts a triple of functions, each of which is polymorphic in the node’s type but monomorphic in its shape:

\[
\text{mkTransfer} \ :: \ (n \to f \to \text{Fact} \ f) \to (n \to f \to \text{Fact} \ f) \to (n \to f \to \text{Fact} \ f) \to \text{FwdTransfer} \ n \ f
\]

We have used this interface to write a number of functions that are polymorphic in the node type \( n \):

- A function that takes a \( \text{FwdTransfer} \) and wraps it in logging code, so an analysis can be debugged by watching facts flow through nodes.
- A pairing function that runs two passes interleaved, not sequentially, potentially producing better results than any sequence:
  \[
  \text{pairFwd} :: \text{forall} \ m \ f \ f'. \ \text{Monad} \ m \to \text{FwdPass} \ m \ n \ f \to \text{FwdPass} \ m \ n \ f' \to \text{FwdPass} \ m \ n \ (f, f')
  \]
- An efficient dominator analysis in the style of Cooper, Harvey, and Kennedy (2001), whose transfer function is implemented using only the functions in the \( \text{NonLocal} \) type class.
4.6 Example: Constant propagation and constant folding

Figure 5 shows client code for constant propagation and constant folding. For each variable, at each program point, the analysis concludes one of three facts: the variable holds a constant value of type \(\text{Lit}\), the variable might hold a non-constant value, or what the variable holds is unknown. We represent these facts using a finite map from a variable to a fact of type \(\text{WithTop Lit}\) (Section 4.4).

A variable with a constant value maps \(\text{Just (PElem k)}\), where \(k\) is the constant value; a variable with a non-constant value maps \(\text{Just Top}\); and a variable with an unknown value maps \(\text{Nothing}\) (it is not in the domain of the finite map).

The definition of the lattice \(\text{constLattice}()\) is straightforward. The bottom element is an empty map (nothing is known about what any variable holds). The join function is implemented with the help of combinators provided by Hoopl. The client writes a simple function, \(\text{constFactAdd}\), which compares two values of type \(\text{Lit}\) and returns a result of type \(\text{withTop Lit}\). The client uses \(\text{extendJoinDomain}\) to lift \(\text{constFactAdd}\) into a join function on \(\text{WithTop Lit}\). Then, \(\text{uses joinMaps}\) to lift \(\text{that join function up to the map containing facts for all variables.}\)

The forward transfer function \(\text{varHasLit}\) is defined using the shape-polymorphic auxiliary function \(ft\). For most nodes \(n\). \(ft\) simply propagates the input fact forward. But for an assignment node, if a variable \(x\) gets a constant value \(k\), \(ft\) extends the input fact by mapping \(x\) to \(\text{PElem k}\). And if a variable \(x\) is assigned a non-constant value, \(ft\) extends the input fact by mapping \(x\) to \(\text{Top}\). There is one other interesting case: a conditional branch where the condition is a variable. If the conditional branch flows to the true successor, the variable holds \(\text{True}\), and similarly for the false successor, \(\text{mutatis mutandis}\). Function \(ft\) updates the fact flowing to each successor accordingly. Because \(ft\) scrutinizes a GADT, it cannot use a wildcard to default the uninteresting cases.

The transfer function need not 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 very simple transfer functions without missing opportunities to improve the code.

Figure 5's rewrite function for constant propagation, \(\text{constProp}\), rewrites each use of a variable to its constant value. The client has defined auxiliary functions that may change expressions or nodes: \(\text{type MaybeChange a = a -> Maybe a}\). mapVE :: (Var -> Maybe Expr) -> MaybeChange Expr mapEN :: MaybeChange Expr -> MaybeChange Expr.

The client composes mapXX functions to apply \(\text{lookup}\) to each use of a variable in each kind of node; \(\text{lookup}\) substitutes for each variable that has a constant value. Applying \(\text{liftM nodeToG}\) lifts the final node, if present, into a \(\text{Graph}\).

Figure 5 also gives another, distinct function for constant folding: \(\text{simp}\). This function rewrites constant expressions to their values, and it rewrites a conditional branch on a boolean constant to an unconditional branch. To rewrite constant expressions, it runs \(\text{s_exp}\) on every subexpression. Function \(\text{simp}\) does not check whether a variable holds a constant value; it relies on \(\text{constProp}\) to have replaced the variable by the constant. Indeed, \(\text{simp}\) does not consult the incoming fact, so it is polymorphic in \(f\).

The \(\text{FwdRewrite}\) functions \(\text{constProp}\) and \(\text{simp}\) are useful independently. In this case, however, we want \(\text{both}\) of them, so we compose them with \(\text{thenFwdRu}\). The composition, along with the lattice and the transfer function, goes into \(\text{constPropPass}\) (bottom of Figure 5). Given \(\text{constPropPass}\), we can improve a graph \(g\) by passing \(\text{constPropPass}\) and \(g\) to \(\text{analyzedAndRewriteFwdBody}\).
4.7 Checkpointing the client’s monad

When analyzing a program with loops, a rewrite function could make a change that later has to be rolled back. For example, consider constant propagation in this loop, which computes factorial:

\[
\begin{align*}
& i = 1; \text{ prod } = 1; \\
& \text{ L1: } \text{ if } (i \geq n) \text{ goto L3 else goto L2;} \\
& \text{ L2: } i = i + 1; \text{ prod } = \text{ prod } \times i; \\
& \quad \quad \text{ goto L1;} \\
& \text{ L3: } \ldots
\end{align*}
\]

Function \texttt{analyzeAndRewriteFwdBody} iterates through this graph until the dataflow facts stop changing. On the first iteration, the assignment \( i = 1 + 1 \) is analyzed with an incoming fact \( i=1 \), and the assignment is rewritten to the graph \( i=2 \). But on a later iteration, the incoming fact increases to \( i=7 \), and the rewrite is no longer justified. After each iteration, \texttt{Hoopl} starts the next iteration with new facts but with the original graph—by virtue of using purely functional data structures, rewrites from previous iterations are automatically rolled back.

But a rewrite function doesn’t only produce new graphs; it can also take monadic actions, such as acquiring a fresh name. These actions must also be rolled back, and because the client chooses the monad in which the actions take place, the client must provide the means to roll back the actions. \texttt{Hoopl} therefore defines a rollback interface, which each client must implement; it is the type class \texttt{CkpointMonad} from Figure 4.

\begin{verbatim}
class Monad m => CkpointMonad m where
  checkpoint :: m (Checkpoint m)
  restart :: Checkpoint m -> m ()
\end{verbatim}

\texttt{Hoopl} calls the checkpoint method at the beginning of an iteration, then calls the \texttt{restart} method if another iteration is necessary. These operations must obey the following algebraic law:

\[
\text{do \{ s <- checkpoint; m; restart s \} == return ()}
\]

where \( m \) represents any combination of monadic actions that might be taken by rewrite functions. (The safest course is to make sure the law holds for any action in the monad.) The type of the saved checkpoint \( s \) is up to the client; it is specified as an associated type of the \texttt{CkpointMonad} class.

4.8 Correctness

Facts computed by the transfer function 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 P0PL paper (Lerner, Grove, and Chambers 2002), here we merely state the conditions for correctness as applied to \texttt{Hoopl}:

- The lattice must have no infinite ascending chains; that is, every sequence of calls to \texttt{fact_join} must eventually return \texttt{NoChange}.
- The transfer function must be monotonic: given a more informative fact in, it must 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 \( f \). Moreover, analysis of \( g \) must produce output fact(s) that are at least as informative as the fact(s) produced by applying the transfer function to \( n \). For example, if the transfer function says that \( x=7 \) after the node \( n \), then after analysis of \( g, x \) had better still be 7.
- A transformation that uses deep rewriting must not return a replacement graph which contains a node that could be rewritten indefinitely.

Under these conditions, the algorithm terminates and is sound.

5. Hoopl’s implementation

Section 4 gives a client’s-eye view of \texttt{Hoopl}, showing how to create analyses and transformations. \texttt{Hoopl}’s interface is simple, but the implementation of interleaved analysis and rewriting is not. 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 a new implementation. It is elegant and short (about a third of the size of our last attempt), and it offers strong compile-time guarantees about shapes. We describe only the implementation of forward analysis and transformation. The implementations of backward analysis and transformation are exactly analogous and are included in \texttt{Hoopl}.

We also explain, in Section 5.5, how we isolate errors in faulty optimizers, and how the fault-isolation machinery is integrated with the rest of the implementation.

5.1 Overview

Instead of the interface function \texttt{analyzeAndRewriteFwdBody}, we present the more polymorphic, private function \texttt{arfGraph}, which is short for “analyze and rewrite forward graph:”

\begin{verbatim}
arfGraph :: forall m n f e x. (CkpointMonad m, NonLocal n)
  -> FwdPass m n f -- lattice, transfers, rewrites
  -> MaybeC e [Label] -- entry points for a closed graph
  -> Graph n e x -- the original graph
  -> Fact e f -- fact(s) flowing into entry/entries
  -> m (DG n e x, Fact x f)
\end{verbatim}

Function \texttt{arfGraph} has a more general type than the function \texttt{analyzeAndRewriteFwdBody} because \texttt{arfGraph} is used recursively to analyze graphs of all shapes. If a graph is closed on entry, a list of entry points must be provided; if the graph is open on entry, the graph’s entry sequence must be the only entry point. The graph’s shape on entry also determines the type of fact or facts flowing in. Finally, the result is a “decorated graph” \( DG f n e x \), and if the graph is open on exit, an “exit fact” flowing out.

A “decorated graph” is one in which each block is decorated with the fact that holds at the start of the block. \( DG \) actually shares a representation with \texttt{Graph}, which is possible because the definition of \texttt{Graph} in Figure 2 contains a white lie: Graph is a type synonym for an underlying type \texttt{Graph'1}, which takes the type of block as an additional parameter. (Similarly, function \texttt{gSplice} in Section 3.4 is actually a higher-order function that takes a block-concatenation function as a parameter.) The truth about \texttt{Graph} and \texttt{DG} is as follows:

\begin{verbatim}
type Graph = Block
  type DG f = Graph' (DBlock f)
data DBlock f n e x = DBlock f (Block n e x)
\end{verbatim}

Type \texttt{DG} is internal to \texttt{Hoopl}; it is not seen by any client. To convert a \texttt{DG} to the \texttt{Graph} and \texttt{FactBase} that are returned by the API function \texttt{analyzeAndRewriteFwdBody}, we use a 12-line function:

\begin{verbatim}
normalizeGraph :: NonLocal n => DG f n e x -> (Graph n e x, FactBase f)
\end{verbatim}
Function arfGraph is implemented as follows:

```
arfGraph pass entries = graph
where
  node :: forall e x . (ShapeLifter e x)
    -> n e x -> f -> m (DG f n e x, Fact x f)
  block :: forall e x .
    Block n e x -> f -> m (DG f n e x, Fact x f)
  body :: [Label] -> LabelMap (Block n C C)
    -> Fact C f -> m (DG f n C C, Fact C f)
  graph :: Graph n e x -> Fact e f -> m (DG f n e x, Fact x f)

... definitions of 'node', 'block', 'body', and 'graph' ...
```

The four auxiliary functions help us separate concerns: for example, only node knows about rewrite functions, and only body knows about fixed points. Each auxiliary function works the same way; it takes a “thing” and returns an extended fact transformer. An extended fact transformer takes dataflow fact(s) coming into the “thing,” and it returns an output fact. It also returns a decorated graph representing the (possibly rewritten) “thing”—that’s the extended part. Finally, because rewrites are monadic, every extended fact transformer is monadic.

The types of the extended fact transformers are not quite identical:

- Extended fact transformers for nodes and blocks have the same type; like forward transfer functions, they expect a fact f rather than the more general Fact e f required for a graph. Because a node or a block has exactly one fact flowing into the entry, it is easiest simply to pass that fact.
- Extended fact transformers for graphs have the most general type, as expressed using Fact: if the graph is open on entry, its fact transformer expects a single fact; if the graph is closed on entry, its fact transformer expects a FactBase.
- Extended fact transformers for bodies have the same type as extended fact transformers for closed/closed graphs.

Function arfGraph and its four auxiliary functions comprise a cycle of mutual recursion: arfGraph calls graph; graph calls body and block; body calls block; block calls node; and node calls arfGraph. These five functions do three different kinds of work: compose extended fact transformers, analyze and rewrite nodes, and compute fixed points.

### 5.2 Analyzing blocks and graphs by composing extended fact transformers

Extended fact transformers compose nicely. For example, block is implemented thus:

```
block :: forall e x .
  Block n e x -> f -> m (DG f n e x, Fact x f)
block (BFIRST n) = node n
block (BFirst n) = node n
block (BLast n) = node n
block (Bcat b1 b2) = block b1 `cat` block b2
```

The composition function cat feeds facts from one extended fact transformer to another, and it splices decorated graphs.

```
cat :: forall e a x f1 f2 f3.
  (f1 -> m (DG f n e a, f2))
  -> (f2 -> m (DG f n a x, f3))
  -> (f1 -> m (DG f n e x, f3))

cat f1 f2 f = do { (g1,f1) <- f1 ; (g2,f2) <- f2 ; return (g1, 'dgSplice' g2, f2) }
```

(Function dgSplice is the same splicing function used for an ordinary Graph, but it uses a one-line block-concatenation function suitable for DBlocks.) The name cat comes from the concatenation of the decorated graphs, but it is also appropriate because the style in which it is used is reminiscent of concatMap, with the node and block functions playing the role of map.

### 5.3 Analyzing and rewriting nodes

The node function is where we interleave analysis with rewriting:

```
node :: forall e x . (ShapeLifter e x)
  -> n e x -> f -> m (DG f n e x, Fact x f)
node n f = do { grw <- frewrite pass n f
  case grw of
    Nothing -> return ( singletonDG f n
                    , tftransfer pass n f )
    Just (g, rw) ->
      let pass' = pass { fp_rewrite = rw }
      in arfGraph pass' (fwdEntryLabel n g f')
```

Function node uses frewrite to extract the rewrite function from pass, and it applies that rewrite function to node n and incoming fact f. The result, grw, is scrutinized by the case expression.

In the Nothing case, no rewrite takes place. We return node n and its incoming fact f as the decorated graph singletonDG f n. To produce the outgoing fact, we apply the transfer function tftransfer pass to n and f.

In the Just case, we receive a replacement graph g and a new rewrite function rw, as specified by the model in Section 4.4. We use rw to analyze and rewrite g recursively with arfGraph. The recursive analysis uses a new pass pass’, which contains the original lattice and transfer function from pass, together with rw.

Function fwdEntryFact converts fact f from the type f, which node has, to the type Fact e f, which arfGraph expects.

As shown above, several functions called in node are overloaded over a (private) class ShapeLifter. Their implementations depend on the open/closed shape of the node. By design, the shape of a node is known statically everywhere node is called, so this use of ShapeLifter is specialized away by the compiler.

### 5.4 Fixed points

The fourth and final auxiliary function of arfGraph is body, which iterates to a fixed point. This part of the implementation is the only really tricky part, and it is cleanly separated from everything else:

```
body :: [Label] -> LabelMap (Block n C C)
  -> Fact C f -> m (DG f n C C, Fact C f)
body entries blockmap init_fbase
  = fixpoint Fwd lattice do_block blocks init_fbase
    where
    blocks = forwardBlockList entries blockmap
    lattice = fp_lattice pass
    do_block b fb = block b entryFact
    where entryFact = getFact lattice (entryLabel b) fb
```

Function getFact looks up a fact by its label. If the label is not found, getFact returns the bottom element of the lattice:

```
getFact :: DataflowLattice f -> Label -> FactBase f -> f
```
Function `forwardBlockList` takes a list of possible entry points and a finite map from labels to blocks. It returns a list of blocks, sorted into an order that makes forward dataflow efficient:

```plaintext
def forwardBlockList :: forall m n f. (CkpointMonad m, NonLocal n) => Direction 
    -> DataflowLattice f 
    -> (Block n C C) 
    -> (Block n C C) 
    -> (Fact C f -> m (DG f n C C, Fact C f)) 
    -> [Block n C C] 
    -> (Fact C f -> m (DG f n C C, Fact C f)) 
```

For example, if the entry point is at L2, and the block at 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 `NonLocal` class—entryLabel and successors—to perform a reverse postorder depth-first traversal of the control-flow graph.

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

```plaintext
def data Direction = Fwd | Bwd 
fixpoint :: forall m n f. (CkpointMonad m, NonLocal n) -> Direction 
    -> DataflowLattice f 
    -> (Block n C C) 
    -> (Fact C f -> m (DG f n C C, Fact C f)) 
    -> [Block n C C] 
    -> (Fact C f -> m (DG f n C C, Fact C f)) 
```

Except for the `Direction` passed as the first argument, the type signature tells the story. The third argument can produce an extended fact transformer for any single block; `fixpoint` applies it successively to each block in the list passed as the fourth argument. Function `fixpoint` returns an extended fact transformer for the list.

The extended fact transformer returned by `fixpoint` maintains a "current FactBase" which grows monotonically: as each block is analyzed, the block's input fact is taken from the current FactBase, and the current FactBase is augmented with the facts that flow out of the block. The initial value of the current FactBase is the input FactBase, and the extended fact transformer iterates over the blocks until the current FactBase stops changing.

Implementing `fixpoint` requires about 90 lines, formatted for narrow display. The code, which is appended to the Web version of this paper (http://bit.ly/cZ7ts1), is mostly straightforward—although we try to be clever about deciding when a new fact means that another iteration is 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:

1. L1: x:=3; goto L4
2. L2: x:=4; goto L4
3. 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 \(x=\top\). Given \(x=\top\), the conditional in L4 cannot be rewritten, and L2 seems reachable. We have lost a good optimization.

Function `fixpoint` solves this problem by analyzing a block only if the block is reachable from an entry point. This trick is safe only for a forward analysis, which is why `fixpoint` takes a `Direction` as its first argument.

### 5.5 Throttling rewriting using "optimization fuel"

When optimization produces a faulty program, we use Whalley’s (1999) technique to find the fault: given a program that fails when compiled with optimization, a binary search on the number of rewrites finds an \(n\) such that the program works after \(n-1\) rewrites but fails after \(n\) rewrites. The \(n\)th rewrite is faulty. As alluded to at the end of Section 6, this technique enables us to debug complex optimizations by identifying one single rewrite that is faulty.

To use this debugging technique, we must be able to control the number of rewrites. We limit rewrites using optimization fuel. Each rewrite consumes one unit of fuel, and when fuel is exhausted, all rewrite functions return `Nothing`. To debug, we do binary search on the amount of fuel.

The supply of fuel is encapsulated in the `FuelMonad` type class (Figure 4), which must be implemented by the client’s monad `m`.

To ensure that each rewrite consumes one unit of fuel, `mkFRewrite` wraps the client’s rewrite function, which must be oblivious to fuel, in another function that satisfies the following contract:

- If the fuel supply is empty, the wrapped function always returns `Nothing`.
- If the wrapped function returns `Just g`, it has the monadic effect of reducing the fuel supply by one unit.

### 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). 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. Steffen (1999) shows that a dataflow analysis can be implemented using model checking. Schmidt (1998) expands on this result by showing that an all-paths dataflow problem can be viewed as model checking an abstract interpretation.

Marlowe and Ryder (1990) present a survey of different methods for performing dataflow analyses, with emphasis on theoretical results. Muchnick (1997) presents many examples of both particular analyses and related algorithms.

Lerner, Grove, and Chambers (2002) show that interleaving analysis and transformation is sound, even when not all speculative transformations are performed on later iterations.

#### 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 frameworks that do interleave analysis and transformation.

The Soot framework is designed for analysis of Java programs (Vallee-Rai et al. 2000). While Soot’s dataflow library supports only analysis, not transformation, we found much to admire in its

---

1 Kildall uses meets, not joins. Lattice orientation is a matter of convention, and conventions have changed. We use Dana Scott’s orientation, in which higher elements carry more information.
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.

The CIL toolkit (Necula et al. 2002) supports both analysis and rewriting of C programs, but rewriting is clearly distinct from analysis: one runs an analysis to completion and then rewrites based on the results. The framework is limited to one representation of control-flow graphs and one representation of instructions, both of which are mandated 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 Whirlwind compiler contains the dataflow framework implemented by Lerner, Grove, and Chambers (2003), 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.

7. Performance considerations

Our work on Hoopl is too new for us to be able to say much about performance. It is important to know how well Hoopl performs, but the question is comparative, and there isn’t another library we can compare Hoopl with. For example, Hoopl is not a drop-in replacement for an existing component of GHC; we introduced Hoopl to GHC as part of a major refactoring of GHC’s back end. With Hoopl, GHC seems about 15% slower than the previous GHC, but we don’t know what part of the slowdown, if any, should be attributed to the optimizer. We can say that the costs of using Hoopl seem reasonable; there is no “big performance hit.” And a somewhat similar library, written in an impure functional language, actually improved performance in an apples-to-apples comparison with a library using a mutable control-flow graph (Ramsey and Dias 2005).

Although thorough evaluation of Hoopl’s performance must await future work, we can identify some design decisions that might affect performance.

- In Figure 2 we show a single concatenation operator for blocks. Using this representation, a block of $N$ nodes is represented using $2N − 1$ heap objects. We have also implemented a representation of blocks that include “cons-like” and “snoc-like” constructors; this representation requires only $N + 1$ heap objects. We don’t know how this choice affects performance.

- In Section 5 the body function analyzes and (speculatively) rewrites the body of a control-flow graph, and fixpoint iterates this analysis until it reaches a fixed point. Decorated graphs computed on earlier iterations are thrown away. For each decorated graph of $N$ nodes, at least $2N − 1$ thunks are allocated; they correspond to applications of singletonDG in node and of dgSplice in cat. In an earlier version of Hoopl, this overhead was eliminated by splitting arfGraph into two phases, as in Whirlwind. The single arfGraph is simpler and easier to maintain; we don’t know if the extra thunks matter.

- The representation of a forward-transfer function is private to Hoopl. Two representations are possible: we may store a triple of functions, one for each shape a node may have; or we may store a single, polymorphic function. Hoopl uses triples, because although working with triples makes some code slightly more complex, the costs are straightforward. If we used a single, polymorphic function, we would have to use a shape classifier (supplied by the client) when composing transfer functions. Using a shape classifier would introduce extra case discriminations every time we applied a transfer function or rewrite function to a node. We don’t know how these extra discriminations might affect performance.

In summary, Hoop performs well enough for use in GHC, but there is much we don’t know. We have no evidence that any of the decisions above measurably affects performance—systematic investigation is indicated.

8. Discussion

We built Hoopl in order to combine three good ideas (interleaved analysis and transformation, an applicative control-flow graph, and optimization fuel) in a way that could easily be reused by many compiler writers. To evaluate how well we succeeded, we examine how Hoopl has been used, we examine the API, and we examine the implementation. We also sketch one of the many alternatives we have implemented.

Using Hoopl. As suggested by the constant-propagation example in Figure 5, Hoopl makes it easy to implement many standard dataflow analyses. Students using Hoopl in a class at Tufts were able to implement such optimizations as lazy code motion (Knoop, Ruething, and Steffen 1992) and induction-variable elimination (Cocke and Kennedy 1977) in just a few weeks. Graduate students at Yale and at Portland State have also implemented a variety of optimizations.

Hoopl’s graphs can support optimizations beyond classic dataflow. For example, in GHC, Hoopl’s graphs are used to implement optimization based on control flow, such as eliminating branch chains.

Hoopl is SSA-neutral: although we know of no attempt to use Hoopl to establish or enforce SSA invariants, Hoop makes it easy to include $\phi$-functions in the representation of first nodes, and if a transformation preserves SSA invariants, it will continue to do so when implemented in Hoopl.

Examining the API. We hope that our presentation of the API in Section 4 speaks for itself, but there are a couple of properties worth highlighting. First, it’s a good sign that the API provides many higher-order combinators that make it easier to write client code. We have had space to mention only a few: extendJoinDomain, joinMaps, thenFwdRw, iterFwdRw, deepFwdRw, and pairFwd.

Second, the static encoding of open and closed shapes at compile time worked out well. Shapes may seem like a small refinement, but they helped eliminate a number of bugs from GHC, and we expect them to help other clients too. GADTs are a convenient way to express shapes, and for clients written in Haskell, they are clearly appropriate. If one wished to port Hoopl to a language without GADTs, many of the benefits could be realized by making the shapes phantom types, but without GADTs, pattern matching would be significantly more tedious and error-prone.
Examining the implementation. If you are thinking of adopting Hoopl, you should consider not only whether you like the API, but whether if you had to, you could maintain the implementation. We believe that Section 5 sketches enough to show that Hoopl’s implementation is a clear improvement over previous implementations of similar ideas. By decomposing our implementation into node, block, body, graph, cat, fixpoint, and mkFRewrite, we have cleanly separated multiple concerns: interleaving analysis with rewriting, throttling rewriting using optimization fuel, and computing a fixed point using speculative rewriting. Because of this separation of concerns, we believe our implementation will be easier to maintain than anything that preceded it.

Design alternatives. We have explored many alternatives to the API presented above. While these alternatives are interesting, describing and discussing an interesting alternative seems to take us a half-column or a column of text. Accordingly, we discuss only the single most interesting alternative: keeping the rewrite monad m private instead of allowing the client to define it.

We have implemented an alternative API in which every rewrite function must use a monad mandated by Hoopl. This alternative has advantages: Hoopl implements checkpoint, restart, setFuel, and getFuel, so we can ensure that they are right and that the client cannot misuse them. The downside is that the only actions a rewrite function can take are the actions in the monad(s) mandated by Hoopl. These monads must therefore provide extra actions that a client might need, such as supplying fresh labels for new blocks. Worse, Hoopl can’t possibly anticipate every action a client might want to take. What if a client wanted one set of unique names for labels and a different set for registers? What if, in order to judge the effectiveness of an optimization, a client wanted to log how many rewrites take place, or in what functions they take place? Or what if a client wanted to implement Primitive Redex Speculation (Runciman 2002), a code-improving transformation that can create new function definitions? Hoopl’s predefined monads don’t accommodate any of these actions. By permitting the client to define the monad m, we risk the possibility that the client may implement key operations incorrectly, but we also ensure that Hoopl can support these examples, as well as other examples not yet thought of.

Final remarks. 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 which makes Haskell 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 Haskell was a win: we had to make every input and output explicit, and we had a strong incentive to implement things compositionally. Using Haskell helped us make real improvements in the implementation of some very sophisticated ideas.

Acknowledgments

Brian Huffman and Graham Hutton helped with algebraic laws. Sukyoung Ryu told us about Primitive Redex Speculation. Several anonymous reviewers helped improve the presentation.

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


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 (possible even at the
Haskell Symposium!), these identifiers are listed in Appendix B.

Add :: Operator not shown (but see page 128).
add defined on page 127.
addBlock defined in Figure 2 on page 123.
analyzeAndRewriteFwdBody defined on page 125.
arfGraph defined on page 129.
Asgn defined in Figure 1 on page 123.
b1 let- or λ-bound on page 124.
b2 let- or λ-bound on page 124.
BCat defined in Figure 2 on page 123.
BFirst defined in Figure 2 on page 123.
Binop :: Operator -> Expr -> Expr -> Expr not shown
(but see page 128).
BLast defined in Figure 2 on page 123.
b1k let- or λ-bound on page 126.
blk let- or λ-bound on page 126.
Block defined in Figure 2 on page 123.
block defined on page 120.
blockmap let- or λ-bound on page 120.
blocks let- or λ-bound on page 120.
blockUnion defined in Figure 2 on page 123.
BMiddle defined in Figure 2 on page 123.
body defined on page 130.
Bot defined on page 126.
Branch defined in Figure 1 on page 124.
bs let- or λ-bound on page 124.
bs1 let- or λ-bound on page 124.
bs2 let- or λ-bound on page 124.
Bwd defined on page 131.
C defined in Figure 2 on page 123.
cat defined on page 130.
cha let- or λ-bound on page 135.
cha' let- or λ-bound on page 135.
chas let- or λ-bound on page 135.
ChangeFlag defined in Figure 4 on page 125.
Checkpoint defined on page 129.
checkpoint defined on page 129.
CheckpointMonad defined on page 129.
Cond defined in Figure 2 on page 123.
ConstFact defined in Figure 5 on page 128.
constFactAdd defined in Figure 5 on page 128.
constLattice defined in Figure 5 on page 128.
constProp defined in Figure 5 on page 128.
constPropPass defined in Figure 5 on page 128.
cp let- or λ-bound in Figure 5 on page 128.
DataflowLattice defined in Figure 4 on page 125.
DBlock defined on page 129.
deepFwdRw defined on page 127.
DG defined on page 129.
dgnilC :: DG f n C C not shown (but see page 136).
dgSplice defined on page 130.
Direction defined on page 131.
direction let- or λ-bound on page 136.
do_block let- or λ-bound on page 130.
entries let- or λ-bound on page 130.
entryFact let- or λ-bound on page 130.
entryLabel defined in Figure 2 on page 123.
ex let- or λ-bound in Figure 2 on page 123.
Expr defined on page 123.
extendJoinDomain defined on page 126.
Fact defined in Figure 4 on page 125.
FactBase defined in Figure 4 on page 125.
fact_bot defined in Figure 4 on page 125.
fact_join defined in Figure 4 on page 125.
fb let- or λ-bound on page 130.
FBase let- or λ-bound on page 125.
FBase' let- or λ-bound on page 130.
fixpoint defined on page 131.
fl let- or λ-bound in Figure 5 on page 128.
forwardBlockList defined on page 131.
fp_lattice defined in Figure 4 on page 125.
fp_rewrite defined in Figure 4 on page 125.
fp_transfer defined in Figure 4 on page 125.
FwdRewrite defined in Figure 4 on page 125.
FwdTransfer defined in Figure 4 on page 125.
getFact defined on page 131.
getFuel defined in Figure 4 on page 125.
GMany defined in Figure 2 on page 123.
GNil defined in Figure 2 on page 123.
Graph defined in Figure 2 on page 123.
Graph' defined on page 129.
grw let- or λ-bound on page 130.
gSplice defined on page 124.
GUnit defined in Figure 2 on page 123.
init_fbase let- or λ-bound on page 130.
init_tx let- or λ-bound on page 136.
in_bls let- or λ-bound on page 136.
is_fwd let- or λ-bound on page 136.
Iter defined on page 127.
iterFwdRw defined in Figure 4 on page 125.
join let- or λ-bound on page 133.
joinFun defined in Figure 4 on page 125.
joinMaps defined on page 126.
JustO defined in Figure 4 on page 123.
Label defined in Figure 4 on page 123.
LabelMap defined in Figure 2 on page 123.
LabelSet (a type) not shown (but see page 136).
lat let- or λ-bound on page 135.
lattice let- or λ-bound on page 130.
llbl let- or λ-bound on page 135.
lbls let- or λ-bound on page 135.
lbls' let- or λ-bound on page 136.
LIter defined on page 128.
lookup let- or λ-bound in Figure 5 on page 128.
lookupFact :: FactBase f -> Label -> Maybe f not shown (but see page 136).
looplet- or λ-bound on page 136.
mapDeleteList :: [Label] -> LabelMap a -> LabelMap a
not shown (but see page 136).
mapEE defined on page 128
mapFoldWithKey :: (Label -> a -> b -> b) -> b -> LabelMap a -> b not shown (but see page 136).
mapInsert :: Label -> a -> LabelMap a -> LabelMap a not shown (but see page 136).
mapMember :: Label -> LabelMap a -> Bool not shown (but see page 136).
mapVE defined on page 128
mapVN defined on page 128
MaybeC (a type of kind * -> * -> *) not shown (but see page 123).
MaybeChange defined on page 128
MaybeO defined in Figure 4 on page 125
mg defined on page 126
mk defined on page 126
mkFactBase defined on page 126
mkFTransfer defined in Figure 4 on page 125
mkFTransfer3 defined on page 127
new let- or \( \lambda \)-bound on page 128
NonLocal defined in Figure 2 on page 123
nodeToG defined in Figure 2 on page 123
NothingO defined in Figure 2 on page 123
old defined in Figure 4 on page 125
oldFact defined in Figure 4 on page 125
old_fact let- or \( \lambda \)-bound on page 135
out_facts let- or \( \lambda \)-bound on page 136
pairFd defined on page 127
pass let- or \( \lambda \)-bound on page 130
pass' let- or \( \lambda \)-bound on page 130
PElem defined on page 126
prod defined on page 126
r let- or \( \lambda \)-bound on page 127
res_fact let- or \( \lambda \)-bound on page 135
restart defined on page 129
rew defined on page 127
rewrite defined on page 127
rg let- or \( \lambda \)-bound on page 136
Rw defined on page 124
rw let- or \( \lambda \)-bound on page 124
setEmpty :: LabelSet not shown (but see page 136).
setFuelList :: [Label] -> LabelSet not shown (but see page 136).
setMember :: Label -> LabelSet -> Bool not shown (but see page 136).
setUnion :: Label -> LabelSet -> LabelSet not shown (but see page 136).
s_exp let- or \( \lambda \)-bound in Figure 5 on page 128
ShapeLifter defined on page 130

B. Identifiers defined in Haskell Prelude or a standard library


C. Computation of fixed points

Function updateFact updates the current FactBase and sets the ChangeFlag.

updateFact :: DataflowLattice f -> LabelSet
-> Label -> f -> (ChangeFlag, FactBase f)
-> (ChangeFlag, FactBase f)
updateFact lat lbl new_fact (cha, fbase)
| NoChange <- cha2 = (cha, fbase)
| lbl 'setMember' lbls = (SomeChange, new_fbase)
| otherwise = (cha, new_fbase)
where

\( \text{cha2}, \text{res_fact} \)
= case lookupFact lat fbase of
  Nothing -> (SomeChange, new_fact_debug)
  Just old_fact -> join old_fact
  where join old_fact =
          fact_join lat lb1
          (OldFact old_fact) (NewFact new_fact)
          (\_, new_fact_debug) = join (fact_bot lat)
  new_fbase = mapInsert lb1 res_fact fbase

135
Datatype TxFactBase accumulates facts (and the transformed code) during the fixpoint iteration.

data TxFactBase n f = TxFB { tfb_fbase :: FactBase f , tfb_rg :: DG f n C C -- Transformed blocks , tfb_cha :: ChangeFlag , tfbLbls :: LabelSet }

fixpoint direction lat do_block blocks init_fbase = do { tx_fb <- loop init_fbase ; return (tfb_rg tx_fb , 'mapDeleteList' tfb_fbase tx_fb ) }

-- The successors of the Graph are the the Labels -- for which we have facts and which are *not* in -- the blocks of the graph

where

tagged_blocks = 'map tag blocks
is_fwd = case direction of { Fwd -> True; Bwd -> False }
tag :: NonLocal t => t C C -> ((Label, t C C), [Label])
tag b = ((entryLabel b, b),
  if is_fwd then [entryLabel b]
  else successors b)
-- 'tag' adds the in-labels of the block;
-- see Note [TxFactBase invariants]

let

loop :: FactBase f -> m (TxFactBase n f)
loop fbase = do { s <- checkpoint }

Here are some of the invariants of the TxFactBase used by algorithm:

• The current FactBase, tfb_fbase, increases monotonically.
• During an iteration, tfbLbls is the set of in-labels of all blocks that have been processed so far this sweep, including the block that is currently being processed. It is a subset of the Labels of the original (not transformed) blocks.
• During an iteration, tfb_cha is set to SomeChange if and only if we decide another iteration will be needed. It is set if the fact in tfb_fbase for a block @L@ changes and @L@ is in tfbLbls. (Until a label enters tfbLbls, its fact in tfb_fbase has not been read, hence it cannot affect the outcome.)