An “Integrated Code Generator” for the Glasgow Haskell Compiler

João Dias, Simon Marlow,
Simon Peyton Jones, Norman Ramsey

Harvard University, Microsoft Research,
and Tufts University

Classic Dataflow “Optimization,” Purely Functionally

Norman Ramsey

Microsoft Research and Tufts University

(also João Dias & Simon Peyton Jones)

Functional compiler writers
should care about imperative code

To run FP as native code, I know two choices:

1. **Rewrite terms to functional CPS, ANF; then to machine code**
2. **Rewrite terms to imperative C- -; then to machine code**

Functional compiler writers should care about imperative code

To run FP as native code, I know two choices:

1. Rewrite terms to functional CPS, ANF; then to machine code
2. Rewrite terms to imperative C--; then to machine code

Why an imperative intermediate language?

- Access to 40 years of code improvement
- You'll do it anyway (TIL, Objective Caml, MLton)

Functional compiler writers should care about imperative code

To run FP as native code, I know two choices:

1. Rewrite terms to functional CPS, ANF; then to machine code
2. Rewrite terms to imperative C--; then to machine code

Why an imperative intermediate language?

- Access to 40 years of code improvement
- You'll do it anyway (TIL, Objective Caml, MLton)

Functional-programming ideas ease the pain

Optimization madness can be made sane

Flee the jargon of "dataflow optimization"

- ~~Constant propagation, copy propagation, code motion, rematerialization, strength reduction~~. . .
- ~~Forward and backward dataflow problems~~
- ~~Kill, gen, transfer functions~~
- ~~Iterative dataflow analysis~~

Optimization madness can be made sane

Flee the jargon of “dataflow optimization”

- ~~Constant propagation, copy propagation, code motion, rematerialization, strength reduction~~. . .
- ~~Forward and backward dataflow problems~~
- ~~Kill, gen, transfer functions~~
- ~~Iterative dataflow analysis~~

Instead consider

- Substitution of equals for equals
- Elimination of unused assignments

Optimization madness can be made sane

Flee the jargon of "dataflow optimization"

- ~~**Constant propagation, copy propagation, code motion, rematerialization, strength reduction**~~**. . .**
- ~~**Forward and backward dataflow problems**~~
- ~~**Kill, gen, transfer functions**~~
- ~~**Iterative dataflow analysis**~~

Instead consider

- **Substitution of equals for equals**
- **Elimination of unused assignments**
- **Strongest postcondition, weakest precondition**
- **Iterative computation of fixed point**

(Appeal to your inner semanticist)

Dataflow's roots are in Hoare logic

Assertions attached to points between statements:

```
{ i = 7 }
i := i + 1
{ i = 8 }
```

Code rewriting is supported by assertions

Substitution of equals for equals

```
{ i = 7 }
i := i + 1
{ i = 8 }
```

```
{ i = 7 }
i := 7 + 1
{ i = 8 }
```

"Constant Propagation"

```
{ i = 7 }
i := 8
{ i = 8 }
```

"Constant Folding"

Code rewriting is supported by assertions

Substitution of equals for equals

```
{ i = 7 }
i := i + 1
{ i = 8 }
```

```
{ i = 7 }
i := 7 + 1
{ i = 8 }
```

“Constant Propagation”

```
{ i = 7 }
i := 8
{ i = 8 }
```

“Constant Folding”

(Notice how dumb the logic is)

Finding useful assertions is critical

Example coming up (more expressive logic now):

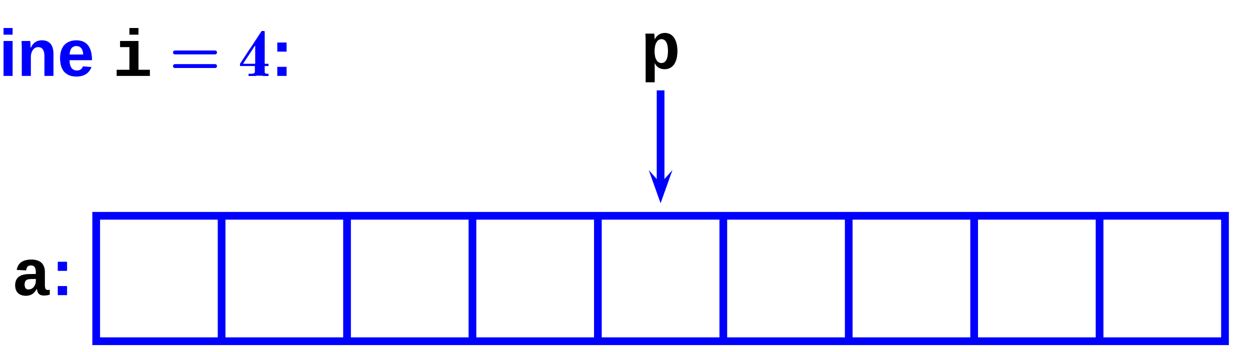
```
{ p = a + i * 12 }
i := i + 1
{ p = a + (i-1) * 12 }
p := p + 12
{ p = a + i * 12 }
```

Dataflow analysis finds good assertions

Example coming up (more expressive logic now):

```
{ p = a + i * 12 }
i := i + 1
{ p = a + (i-1) * 12 }
p := p + 12
{ p = a + i * 12 }
```

Imagine `i` $= 4$:



Example: Classic array optimization

First running example (C code):

```
long double sum(long double a[], int n) {
  long double x = 0.0;
  int i;
  for (i = 0; i < n; i++)
    x += a[i];
  return x;
}
```

Array optimization at machine level

Same example (C-- code):

```
sum("address" bits32 a, bits32 n) {
     bits80 x; bits32 i;
     x = 0.0;
     i = 0;
 L1: if (i >= n) goto L2;
     x = %fadd(x, %f2f80(bits96[a+i*12]));
     i = i + 1;
     goto L1;
 L2: return x;
}
```

Ad-hoc transformation

New variable satisfying `p == a + i * 12`

```
sum("address" bits32 a, bits32 n) {
     bits80 x; bits32 i; bits32 p, lim;
     x = 0.0;
     i = 0; p = a; lim = a + n * 12;
 L1: if (i >= n) goto L2;
     x = %fadd(x, %f2f80(bits96[a+i*12]));
     i = i + 1; p = p + 12;
     goto L1;
 L2: return x;
}
```

“Induction-variable elimination”

Use `p == a + i * 12` and `(i >= n) == (p >= lim)`:

```
sum("address" bits32 a, bits32 n) {
        bits80 x; bits32 i; bits32 p, lim;
        x = 0.0;
        i = 0; p = a; lim = a + n * 12;
  L1: if (p >= lim) goto L2;
        x = %fadd(x, %f2f80(bits96[p]));
        i = i + 1; p = p + 12;
        goto L1;
  L2: return x;
}
```

Finally, `i` is superfluous

"Dead-assignment elimination" (with a twist)

```
sum("address" bits32 a, bits32 n) {
     bits80 x; bits32 i; bits32 p, lim;
     x = 0.0;
     i = 0; p = a; lim = a + n * 12;
 L1: if (p >= lim) goto L2;
     x = %fadd(x, %f2f80(bits96[p]));
     i = i + 1; p = p + 12;
     goto L1;
 L2: return x;
}
```

Finally, `i` is superfluous

“Dead-assignment elimination” (with a twist)

```
sum("address" bits32 a, bits32 n) {
    bits80 x;                   bits32 p, lim;
    x = 0.0;
           p = a; lim = a + n * 12;
 L1: if (p >= lim) goto L2;
    x = %fadd(x, %f2f80(bits96[p]));
               p = p + 12;
    goto L1;
 L2: return x;
}
```

Things we can talk about

Here and now:

- **Example of code improvement ("optimization") grounded in Hoare logic**
- **Closer look at assertions and logic**

Things we can talk about

Here and now:

- Example of code improvement ("optimization") grounded in Hoare logic
- Closer look at assertions and logic

Possible sketches before I yield the floor:

- Ingredients of a "best simple" optimizer
- Bowdlerized code
- Data structures for "imperative optimization" in a functional world

Things we can talk about

Here and now:

- **Example of code improvement ("optimization") grounded in Hoare logic**
- **Closer look at assertions and logic**

Possible sketches before I yield the floor:

- **Ingredients of a "best simple" optimizer**
- **Bowdlerized code**
- **Data structures for "imperative optimization" in a functional world**

Hallway hacking:

- **Real code! In GHC now!**

Assertions and logic

Where do assertions come from?

Key observation:

Statements relate assertions to assertions

Example, Dijkstra's weakest precondition:

$$\boldsymbol{A}_{i-1} = wp(\boldsymbol{S}_i, \boldsymbol{A}_i)$$

(Also good: strongest postcondition)

Query: given $\{S_i\}$, $A_0 = \texttt{True}$, can we solve for $\{A_i\}$?

Answer: Solution exists, but seldom in closed form.

Why not? Disjunction (from loops) ruins everything: fixed point is an infinite term.

Dijkstra's way out: hand write key A's

Dijkstra says: write loop invariant:

An assertion at a join point (loop header)

- May be stronger than necessary
- Can prove verification condition

Dijkstra's way out: hand write key A's

Dijkstra says: write loop invariant:

An assertion at a join point (loop header)

- May be stronger than necessary
- Can prove verification condition

My opinion: a great teaching tool

- Dijkstra/Gries $\equiv$ imperative programming with loops and arrays
- Bird/Wadler $\equiv$ applicative programming with equational reasoning

Dijkstra's way out: hand write key A's

Dijkstra says: write loop invariant:

An assertion at a join point (loop header)

- May be stronger than necessary
- Can prove verification condition

My opinion: a great teaching tool

- Dijkstra/Gries $\equiv$ imperative programming with loops and arrays
- Bird/Wadler $\equiv$ applicative programming with equational reasoning

Not available to compiler

Compiler's way out: less expressive logic

Ultra-simple logics!

(inexpressible predicates abandoned)

Results: weaker assertions at key points

Consequence:

- **Proliferation of inexpressive logics**
- **Each has a name, often a program transformation**
- **Transformation is usually substitution**

Examples:

$P ::= \bot \mid P \wedge x = k$ **"constant propagation"**

$P ::= \bot \mid P \wedge x = y$ **"copy propagation"**

Dataflow analysis solves recursion equations

Easy to think about least solutions:

$A_{i-1} = wp(S_i, A_i), A_{last} = \bot$ **"Backward analysis"**

$A_i = sp(S_i, A_{i-1}), A_0 = \bot$ **"Forward analysis"**

Classic method is iterative, uses mutable state:

1. **Set all** $A_i := \bot$
2. **Repeat for all** i**:**
   let $A'_{i-1} = A_{i-1} \sqcup wp(S_i, A_i)$
   If $A'_{i-1} \neq A_{i-1}$**, set** $A_{i-1} := A'_{i-1}$
3. **Continue until fixed point is reached**

Number of iterations is roughly loop nesting depth

Beyond Hoare logic: The context

Classic assertions are about program state σ

- **Example:** `{ i = 7 }` $\equiv \forall \sigma : \sigma(i) = 7$

Also want to assert about context or continuation θ

- **Example:** `{ dead(x) }` $\equiv \forall \sigma, v : \theta(\sigma) = \theta(\sigma\{x \mapsto v\})$
  (Undecidable, approximate by reachability)
  (Typically track live, not dead)

A “best simple” optimizer for GHC

(Shout if you’d rather see code)

Long-term goal: Haskell, optimized

Classic dataflow-based code improvement, planted in the Glasgow Haskell Compiler (GHC)

Long-term goal: Haskell, optimized

Classic dataflow-based code improvement, planted in the Glasgow Haskell Compiler (GHC)

The engineering question:

- **How to support 40 years of imperative-style analysis and optimization simply, cleanly, and in a purely functional setting?**

Long-term goal: Haskell, optimized

Classic dataflow-based code improvement, planted in the Glasgow Haskell Compiler (GHC)

The engineering question:

- **How to support 40 years of imperative-style analysis and optimization simply, cleanly, and in a purely functional setting?**

Answers:

- **Good data structures**
- **Powerful code-rewriting engine based on dataflow (i.e. Hoare logic)**

Optimization: a closer look

It’s about registers, loops, and arrays

Dataflow-based optimization

- **Not glamorous like equational reasoning, λ-lifting, closure conversion, CPS conversion**
- **Needs to happen anyway, downstream**

It's about registers, loops, and arrays

Dataflow-based optimization

- Not glamorous like equational reasoning, λ-lifting, closure conversion, CPS conversion
- Needs to happen anyway, downstream

Lesson learned: low-level optimization matters

- TIL (Tarditi)
- Objective Caml (Leroy)
- MLton (Weeks, Fluet, . . .)
- GHC?

Simple ingredients can do a lot

You must be able to

- Represent assignments, control flow graphically (at the machine level)
- Have infinitely many registers (or facsimile)
- Implement a few impoverished logics
- Solve recursion equations (dataflow analysis)
- Mutate assignments and branches

We have 5 essential ingredients

We have 5 essential ingredients

Zipper control-flow graph

(Ramsey and Dias 2005)

We have 5 essential ingredients

Dataflow monad

Zipper control-flow graph

(Ramsey and Dias 2005)

We have 5 essential ingredients

Dataflow analysis

Dataflow monad

Zipper control-flow graph
(Ramsey and Dias 2005)

We have 5 essential ingredients

Interleaved analysis and transformation
(Lerner, Grove, and Chambers 2002)

Dataflow analysis

Dataflow monad

Zipper control-flow graph
(Ramsey and Dias 2005)

We have 5 essential ingredients

Interleaved analysis and transformation
(Lerner, Grove, and Chambers 2002)

Dataflow analysis

Dataflow monad

Zipper control-flow graph
(Ramsey and Dias 2005)

. . . and a good register allocator

Design philosophy

The "33-pass compiler"

- **Small, simple, composable transformations**
- **"Existing optimizations clean up after new optimizations"**
- **Keep improving until code doesn't change**

Simple debugging technique wins big!

Limitable supply of “optimization fuel”

- Rewrite for performance consumes one unit
- On failure, binary search on fuel supply (spread over multiple compilation units)

Invented by David Whalley (1994)

Simple debugging technique wins big!

Limitable supply of “optimization fuel”

- Rewrite for performance consumes one unit
- On failure, binary search on fuel supply (spread over multiple compilation units)

Invented by David Whalley (1994)

Bookkeeping in a “fuel monad”

What’s important

Things to remember

Dataflow analysis =

weakest preconditions + impoverished logic

“Optimization” is largely “equals for equals”

“Movement” is achieved in three steps:

1. **Insert new code**
2. **Rewrite code in place**
3. **Delete old code**

The compiler writer has three good friends:

- **Coalescing register allocator**
- **Dataflow-based transformation engine**
- **“Optimization fuel”**

Dataflow (from 10,000 ft)

(Shout if you prefer the zipper)

Lies, damn lies, type signatures

Logical formula is "dataflow fact"

```
data DataflowLattice a = DataflowLattice {
  bottom  :: a,
  join    :: a -> a,
  refines :: a -> a -> bool
}
```

Facts computed by "transfer function" (wp or sp):

```
type Transfer a = a -> Node -> a
```

Fact might justify a rewrite:

```
type Rewrite  a = a -> Node -> Maybe Graph
```

Bigger, more interesting lies

```
solve :: DataflowLattice a
      -> Transfer a
      -> a   -- fact in (at entry or exit)
      -> Graph
      -> BlockEnv a -- FP: {label |-> fact}

rewr  :: DataflowLattice a
      -> Transfer a
      -> a
      -> RewritingDepth
      -> Rewrite a
      -> Graph
      -> FuelMonad (Graph, BlockEnv a)
```

Simple, almost-true client: liveness

Lattice is set of live registers; join is union.

Transfer equations use traditional `gen`, `kill`:

```
gen, kill :: HasRegs a => a -> RegSet -> RegSet
gen  = foldFreeRegs extendRegSet
kill = foldFreeRegs delOneFromRegSet

xfer :: Transfer RegSet
xfer :: Node -> RegSet -> RegSet
xfer (Comment {})       = id
xfer (Load reg expr)    = gen expr . kill reg
xfer (Store addr rval) = gen addr . gen rval
xfer (Call f res args) = gen f . gen args . kill res
xfer (Return e)         = \ _ -> gen e $ emptyRegSet
```

Companion: dead-assignment elimination

Our most useful tool is dirt-simple:

```
removeDeads :: Rewrite RegSet
removeDeads :: RegSet -> Node -> Maybe Graph
removeDeads live (Load reg expr)
              | not (reg ‘elemRegSet‘ live)
              = Just emptyGraph
removeDeads live _ = Nothing
```

Combine with liveness `xfer` **using** `rewr`

Win by isolating complexity

Function `rewr` is scary (= 1 POPL paper)

Clients are simple:

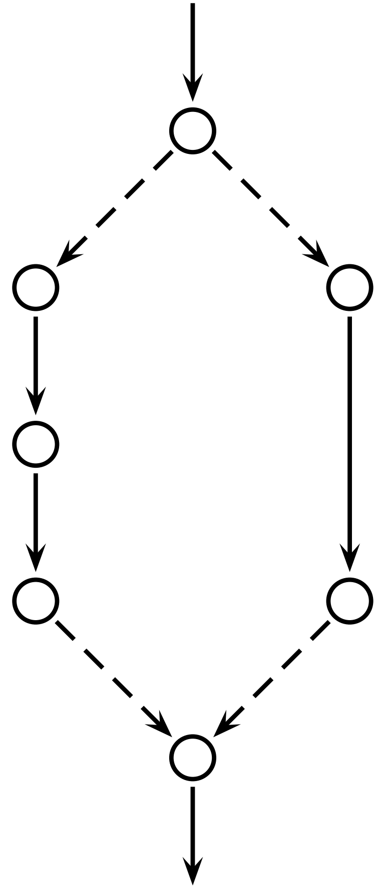
- “Impoverished logic” = “easy to understand”
- Not much code

More examples:

- Spill/reload in 3 passes (1 to insert, 2 to sink)
- Call elimination in 1 pass
- Linear-scan register allocation in 4 passes! (Dias)

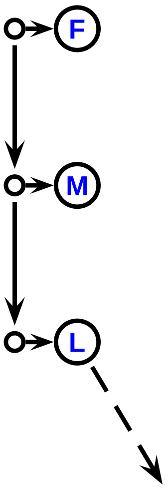
The zipper

A very simple flow graph

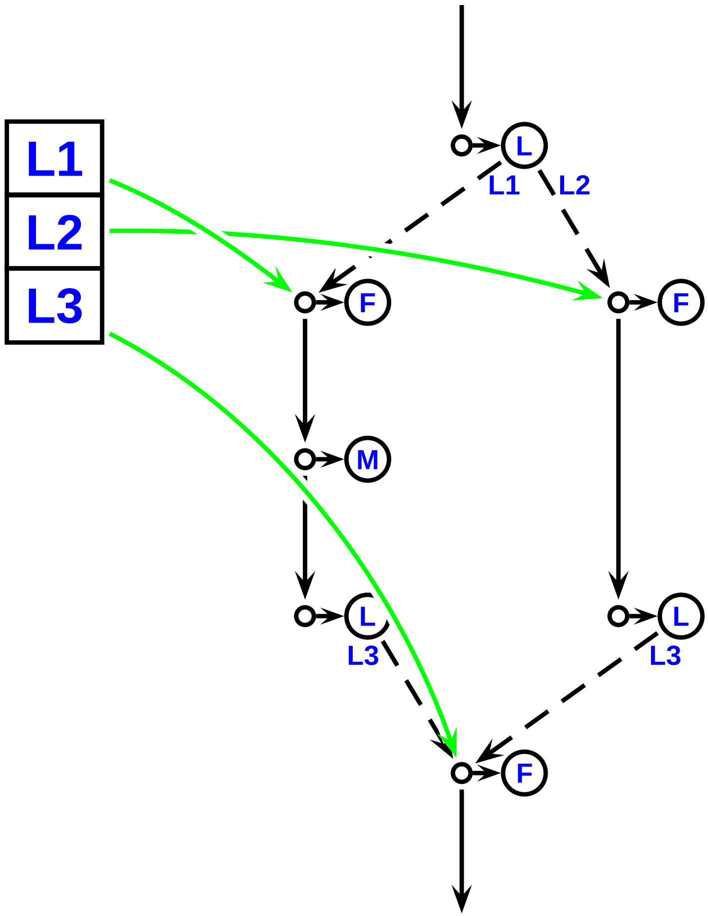


Nodes have different static types

One basic block:



Edges betweeen blocks use a finite map



Need operations on nodes

Not requiring mutation:

- Forward, backward traversal

More imperative-looking:

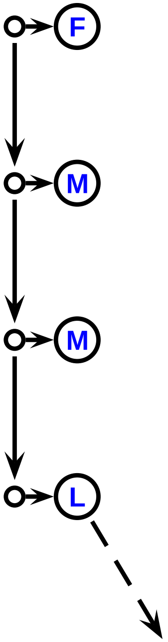
- Insert
- Replace
- Delete

All should be simple, easy, and functional

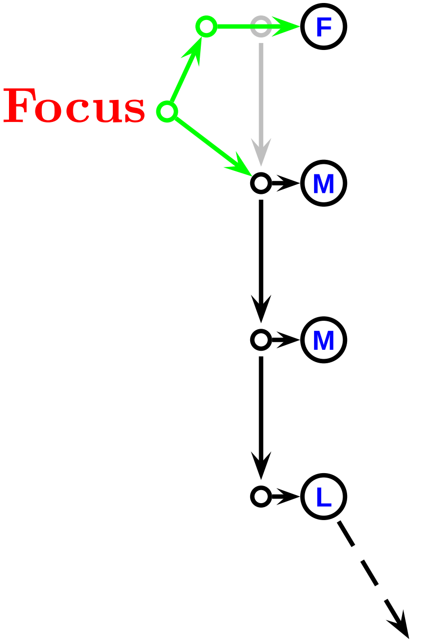
The Zipper: Manipulating basic blocks

The *focus* represents the “current” edge:

Unfocused



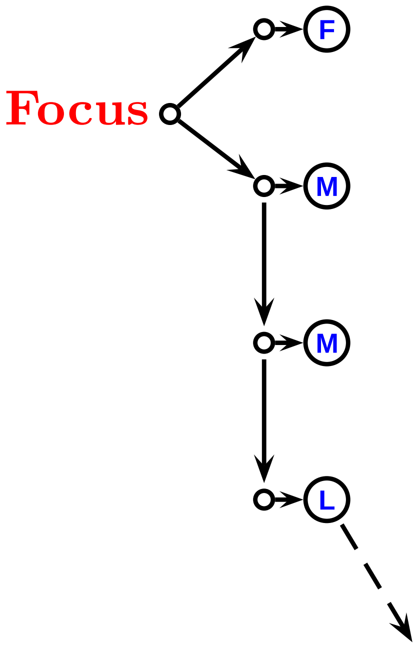
Focused on 1st edge



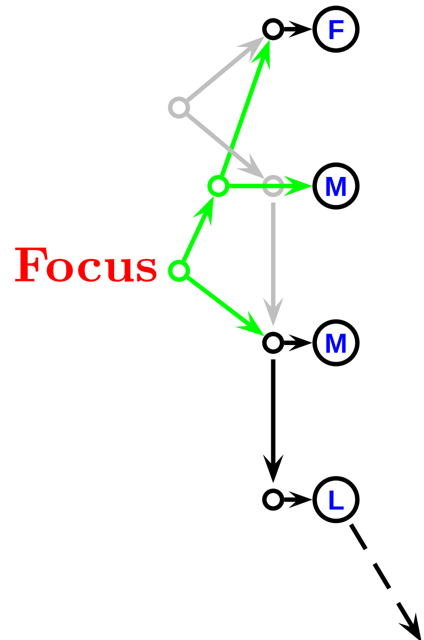
Moving the focus

Traversal requires constant-space allocation:

Focused on 1st edge



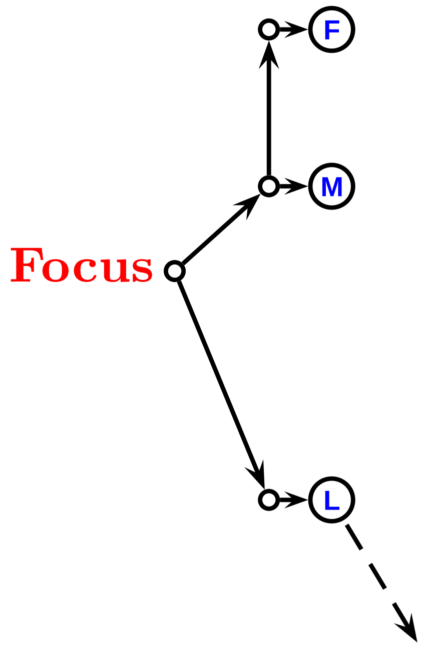
Focused on 2nd edge



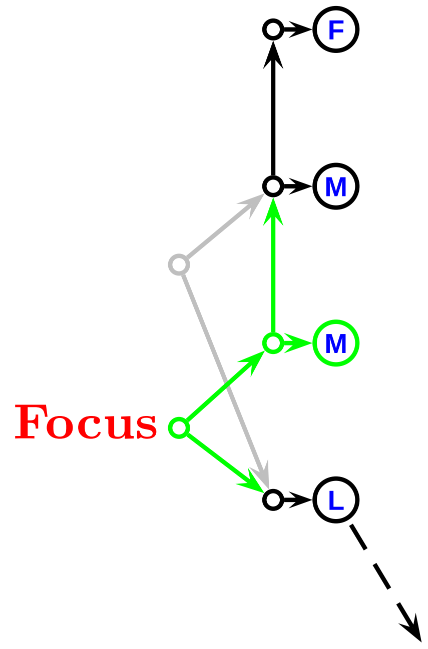
Inserting an instruction

Insertion also requires constant-space allocation:

Focused on 2nd edge



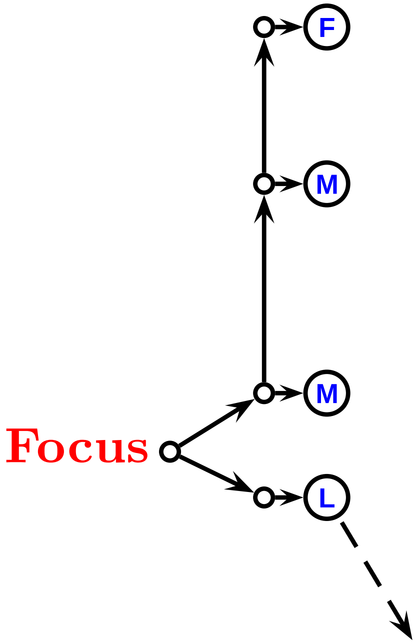
Focused on edge after new instruction



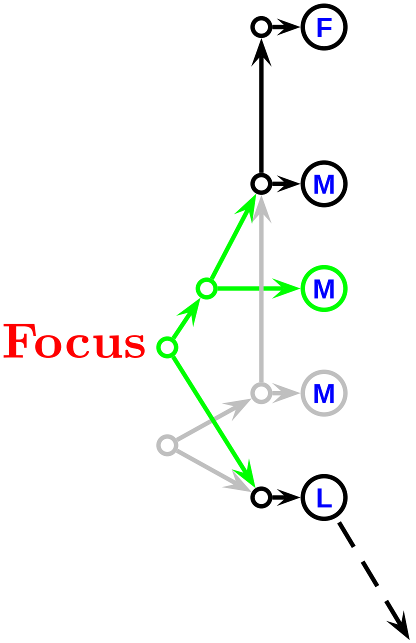
Replacing an instruction

Replacement requires constant-space allocation:

Focused after node to replace



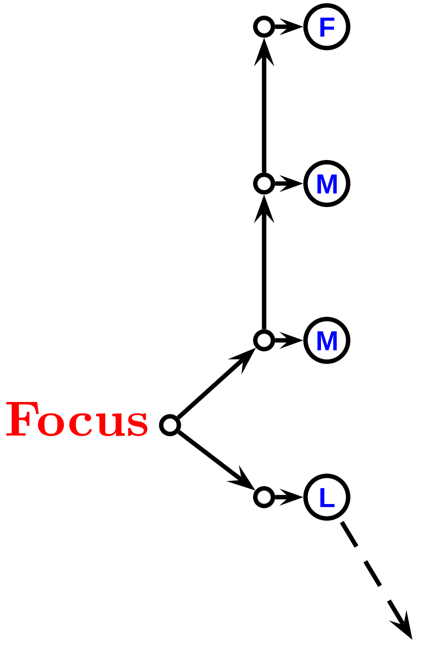
Focused after new node



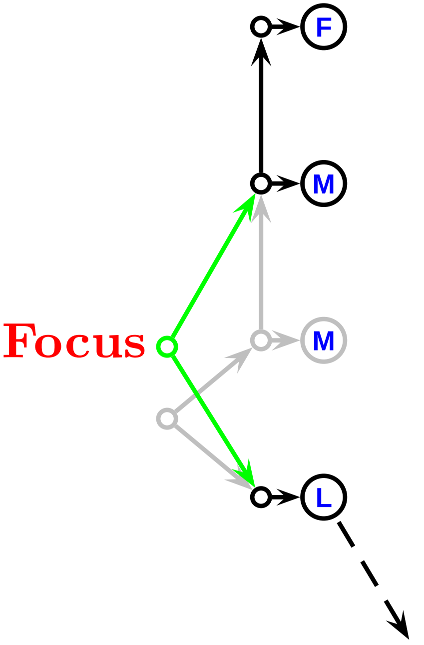
Deleting an instruction

Deletion requires (half) constant-space allocation:

Focused after delendum



Focused on new edge



Benefits of the zipper

Representation with

- **No mutable pointers (or pointer invariants)**
- **Single instruction per node**
- **Easy forward and backward traversal**
- **Incremental update (imperative feel)**

Haskell code

The zipper in Haskell

The “first” node is always a unique identifier

```
data Block m l = Block BlockId (ZTail m l)
data ZTail m l = ZTail m (ZTail m l) | ZLast (ZLast l
  -- sequence of m’s followed by single l
data ZLast l = LastExit | LastOther l
  -- ’fall through’ or a real node
data ZHead m   = ZFirst BlockId  | ZHead (ZHead m) m
  -- (reversed) sequence of m’s preceded by BlockId

data Graph m l =
  Graph (ZTail m l) (BlockEnv (Block m l))
  -- entry sequence paired with collection of blocks

data LGraph m l =
  LGraph BlockId (BlockEnv (Block m l))
  -- for dataflow, every block bears a label
```

Instantiating the zipper

```
data Middle
  = Assign CmmReg CmmExpr    -- Assign to register
  | Store CmmExpr CmmExpr    -- Store to memory
  | UnsafeCall CmmCallTarget CmmResults CmmActuals
                 -- a ’fat machine instruction’
data Last
  = Branch BlockId  -- Goto block in this proc
  | CondBranch {    -- conditional branch
        cml_pred :: CmmExpr,
        cml_true, cml_false :: BlockId
    }
  | Return          -- Function return
  | Jump CmmExpr    -- Tail call
  | Call {          -- Function call
        cml_target :: CmmExpr,
        cml_cont   :: Maybe BlockId }
            -- cml_cont present if call returns
```

Ask me about `CmmSpillReload.hs`

At every `Call` site,

- Every live variable must be saved on the "Haskell stack"

Given: C`--` with local variables live across calls

Produce: C`--` with spills and reloads,
nothing live in a register at any call

(Code produced on demand)

Beyond be dragons

Simple facts might be enough

Transfers, rewrites can compose.

Conjoin facts:

```
(<*>) :: Transfer a -> Transfer b
      -> Transfer (a, b)
```

Sum rewrites:

```
(<+) :: Rewrite a -> Rewrite a -> Rewrite a
```

Rewrite based on conjoined facts:

```
liftR :: (b -> a) -> Rewrite a -> Rewrite b
```