An Applicative Control-Flow Graph Based on Huet’s Zipper

Norman Ramsey and João Dias
{nr, dias}@eecs.harvard.edu

Division of Engineering and Applied Sciences
Harvard University
Optimizing compiler in ML

Compiler = killer app for ML
A functional view of compilation

Compiling $\equiv$ tree rewriting
- Lambda terms
- CPS
- A-normal form

ML shines
- Pattern matching to find subtrees
- Static types unreasonably effective
An imperative view of compilation

Compiling $\equiv$ Dragon book
- Register allocation
- Iterative dataflow analysis
- Peephole optimization
- Irreducible control flow

Iterative mutation of control-flow graph – not trees
Old imperative flow graph

The data structure:

- One instruction per node
- Mutable pointers link nodes (both directions)
- Mutable data at each node and each join point
Old imperative flow graph

The data structure:
- One instruction per node
- Mutable pointers link nodes (both directions)
- Mutable data at each node and each join point

Two main dynamic invariants:
- Successor, predecessor edges match
Old imperative flow graph

The data structure:

- One instruction per node
- Mutable pointers link nodes (both directions)
- Mutable data at each node and each join point

Two main dynamic invariants:

- Successor, predecessor edges match
- Common successor of two nodes must be join point
Invariants are hard to encapsulate

Client maintains invariants
  • every client must know CFG invariants
Invariants are hard to encapsulate

Flow graph maintains invariants
- Hides mutation, invalidating pointers the client holds
- Example: deleting a node

```
a.next := b;
```
Invariants are hard to encapsulate

Flow graph maintains invariants
- Hides mutation, invalidating pointers the client holds
- Example: deleting a node
  ```
a.next := b;
  ```
Invariants are hard to encapsulate

Flow graph maintains invariants
  • Hides mutation, invalidating pointers the client holds
  • Example: deleting a node
    \[ a.\text{next} := b; \]
Imperative CFG was painful

5 major revisions

- First 4 worked only for subset of language
Imperative CFG was painful

5 major revisions
  • First 4 worked only for subset of language

Fear of deep analyses and transformations
Imperative CFG was painful

5 major revisions
  • First 4 worked only for subset of language

Fear of deep analyses and transformations

Did not exploit ML’s strengths
  • too many non-exhaustive patterns
  • not enough static invariants
Looking for applicative alternatives

GHC:
- Basic block has immutable list of instructions
- Each basic block has a unique identifier (uid)
- Finite maps from uid’s to dataflow information

MLton:
- Basic block has immutable vector of instructions
- Mutable dataflow facts
What we want in a CFG

Helpful:

- No mutable pointer invariants (GHC, MLton)
- Single instruction per node (QC--)
- Easy forward, backward traversal (MLton, QC--)
- Incremental update (QC--)
- Mutable dataflow information (MLton)

Not helpful:

- Vectors of nodes (MLton)
- Having no mutable data (GHC)
Key Decision: representing edges

Cycles require indirection

- back edges
- potential back edges

Rediscovered basic blocks

What form of indirection?
- mutable ref cells
- finite maps
Key Decision: representing edges

Cycles require indirection
Which edges indirect?
- back edges
Key Decision: representing edges

Cycles require indirection
Which edges indirect?
- back edges
- potential back edges
Key Decision: representing edges

Cycles require indirection
Which edges indirect?
- back edges
- potential back edges
Rediscovered basic blocks
Key Decision: representing edges

Cycles require indirection
Which edges indirect?
• back edges
• potential back edges
Rediscovered basic blocks
What form of indirection?
• mutable ref cells
• finite maps
Successor edges only

Changes:
1. Successor edges only
Distinguishing between edges

Changes:
1. Successor edges only
2. Intra- vs inter-block edges (basic blocks)
A simplified view of a basic block
Cons cells link straight-line code

Changes:
1. Successor edges only
2. Intra- vs inter-block edges (basic blocks)
3. Cons cells
Nodes have different static types

Changes:
1. Successor edges only
2. Intra- vs inter-block edges (basic blocks)
3. Cons cells
4. 3 static types of nodes
Indirect edges use a lookup table
Operations we want in basic blocks

Easy forward, backward traversal

Incremental update:
  • insert
  • replace
  • delete
The Zipper: Manipulating basic blocks

The *focus* represents the “current” edge:

Unfocussed

Focused on 1st edge

Focus
Moving the focus

Traversing edges by allocating cons cells:

Focused on 1st edge  Focused on 2nd edge
Inserting an instruction

Inserting instruction by allocating cons cells:

Focused on 2nd edge

Focused on edge after new instruction
Replacing an instruction

Replacing an instruction by allocating cons cells:

Focused after node to replace

Focused after new node
Deleting an instruction

Deleting an instruction by allocating a cons cell:

Focused after delendum

Focused on new edge
Benefits of the zipper

Basic blocks support:

- no mutable pointers (or pointer invariants)
- single instruction per node
- easy forward and backward traversal
- incremental update (imperative feel)
How does iterative dataflow work?

Standard textbook approach
- order the basic blocks
- iterate over the ordered list

Dataflow results stored using mutable state

Backward dataflow requires no special support despite lack of back edges
Tastes great

Control-flow graph
  • implementation is shorter (425 lines vs 1400)
  • no fear of modification

Performance slightly better
Clients also win

Client code is simpler
  • first / middle / last nodes help
  • static knowledge of control-flow edges
  • no fear of breaking invariants
  • implementations have imperative flavor
Clients also win

Client code is simpler
- first / middle / last nodes help
- static knowledge of control-flow edges
- no fear of breaking invariants
- implementations have imperative flavor

Dataflow analysis
- mutable dataflow per block a good tradeoff
- composed analyses & transformations are easy
- persistence makes speculative update easy
What we (re)learned

Even where imperative code seems obvious, applicative code can make sense

• Try the zipper