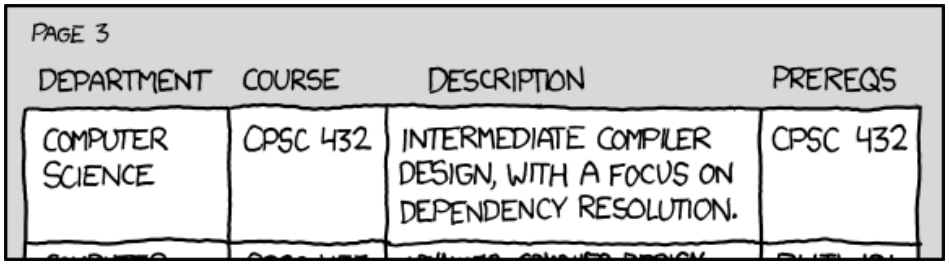
Global Value Numbering in Factor

Alex Vondrak

ajvondrak@csupomona.edu

September 1, 2011



Factor

Factor (`http://factorcode.org/`)

- Started development September 2003—a baby among languages
- Stack-based
- Object-oriented
- Dynamically typed
- Extensive standard library
- High-level, yet fully compiled

Won't really have time to discuss the language in depth

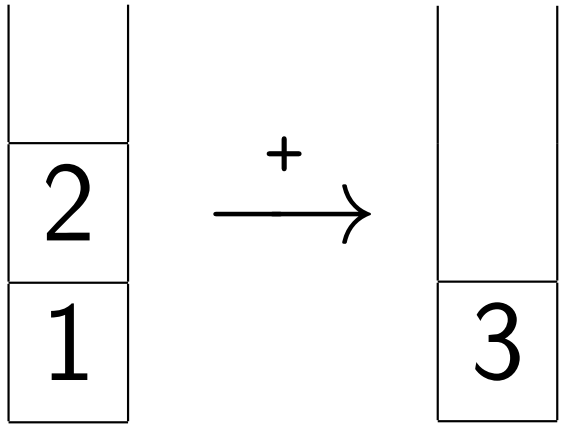
Stacks as an Evaluation Model

Example (Code)

```
1 2 +
```

Example (Execution)

```
push(1);
push(2);
y = pop();    // y = 2;
x = pop();    // x = 1;
push(x + y); // push(3);
```



Organization

Non-optimizing base compiler

- VM written in C++
- Responsible for basic runtime services
  - Garbage collection
  - Method dispatch
  - Polymorphic inline caches
  - ...
- Single pass—outputs assembly stubs for primitives

Optimizing compiler

- Written in Factor code
  - Possible by *bootstrapping*
- Optimizes in passes across two intermediate representations (IRs)
  - High-level IR (`compiler.tree`)
  - Low-level IR (`compiler.cfg`)

High-level IR

- Tree of `node` objects
- Very simple virtual instruction set
  - `#introduce`, `#return`
  - `#push` & `#call`
  - `#renaming`—`#copy` & `#shuffle`
  - `#declare` & `#terminate`
  - `#branch`—`#if` & `#dispatch`
  - `#phi`
  - `#recursive`, `#enter-recursive`, `#call-recursive`, `#return-recursive`
  - `#alien-node`, `#alien-invoke`, `#alien-indirect`, `#alien-assembly`, `#alien-callback`
- Input/output values of stack given unique names

High-level IR

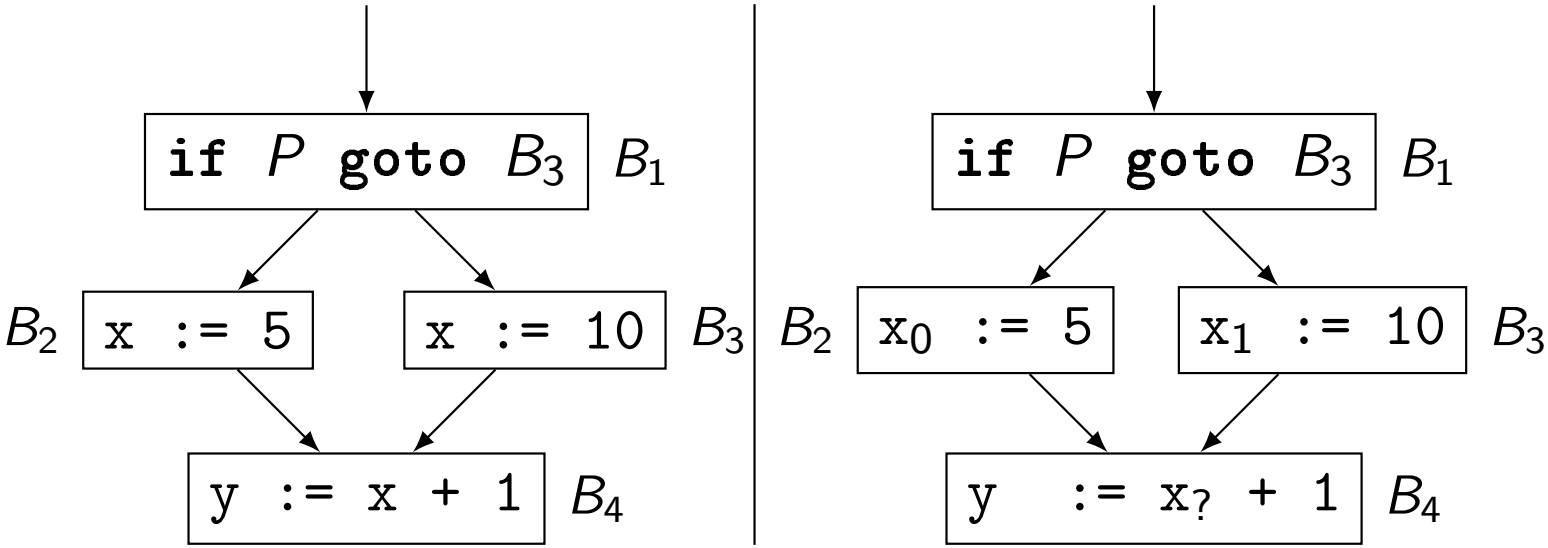
1 2 +

Example

```
V{
    T{ #push { literal 1 } { out-d { 6256273 } } }
    T{ #push { literal 2 } { out-d { 6256274 } } }
    T{ #call
        { word + }
        { in-d V{ 6256273 6256274 } }
        { out-d { 6256275 } }
    }
    T{ #return { in-d V{ 6256275 } } }
}
```

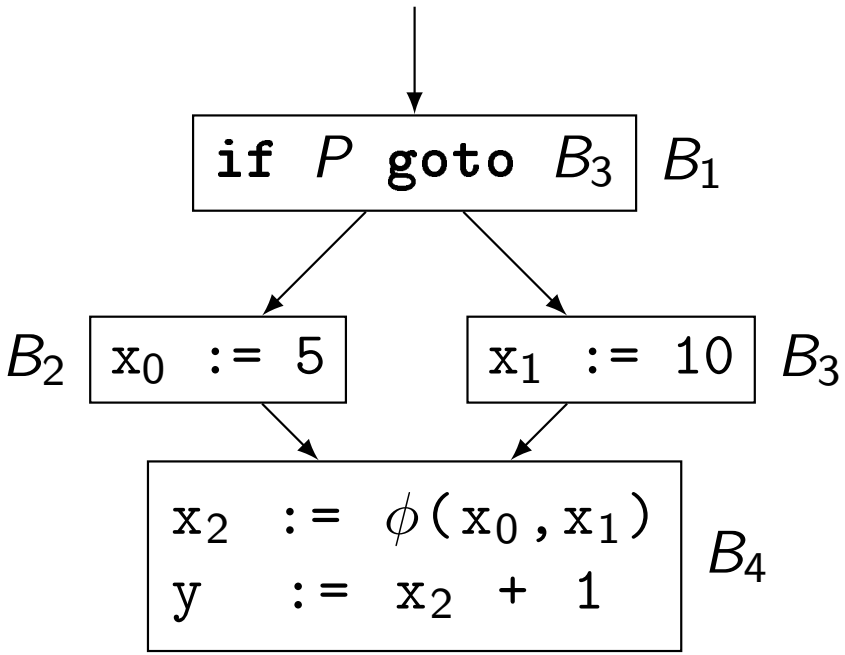
Low-level IR

- Control flow graph (CFG)
  - Basic blocks = maximal sequence of "straight-line" code
  - Directed edges = transfer of control flow
- `insn` objects modeled closely after assembly-like instructions
- Static single assignment (SSA) form



Low-level IR

- Control flow graph (CFG)
  - Basic blocks = maximal sequence of "straight-line" code
  - Directed edges = transfer of control flow
- `insn` objects modeled closely after assembly-like instructions
- Static single assignment (SSA) form



Optimizations—High-level IR

```
: optimize-tree ( nodes -- nodes' )
  [
      analyze-recursive
      normalize
      propagate
      cleanup
      dup run-escape-analysis? [
          escape-analysis
          unbox-tuples
      ] when
      apply-identities
      compute-def-use
      remove-dead-code
      ?check
      compute-def-use
      optimize-modular-arithmetic
      finalize
  ] with-scope ;
```

Optimizations—Low-level IR

```
: optimize-cfg ( cfg -- cfg' )
    optimize-tail-calls
    delete-useless-conditionals
    split-branches
    join-blocks
    normalize-height
    construct-ssa
    alias-analysis
→   value-numbering
    copy-propagation
    eliminate-dead-code ;
```

Value Numbering

Idea: assign each variable a value number

- Equal value numbers $\Longrightarrow$ equal at runtime
- Turn recomputations into ##copy instructions, saving time

General problem is undecidable

- Seek conservative solution
- Discover *Herbrand equivalences*
- Consider two values congruent if
  - They're computed by the same operator
  - Their operands are congruent

Local Value Numbering

- Thought to be invented by Balke in the 1960s
- Largely credited to Cocke & Schwartz in the 1970s
- Current implementation Factor uses

Pro: Easy to understand, implement, and extend

Con: Is local and pessimistic, discovering fewer congruences

Local Value Numbering

Implementation

- Expressions are constructed from instructions

Example

```
T{ ##add { dst 1 } { src1 2 } { src2 3 } } >expr
{ ##add 2 3 }
```

- Expression graph = 3 global hash tables
  - `vregs>vns`
  - `exprs>vns`
  - `vns>insns`
- If possible, instructions are simplified using data from expression graph

Local Value Numbering

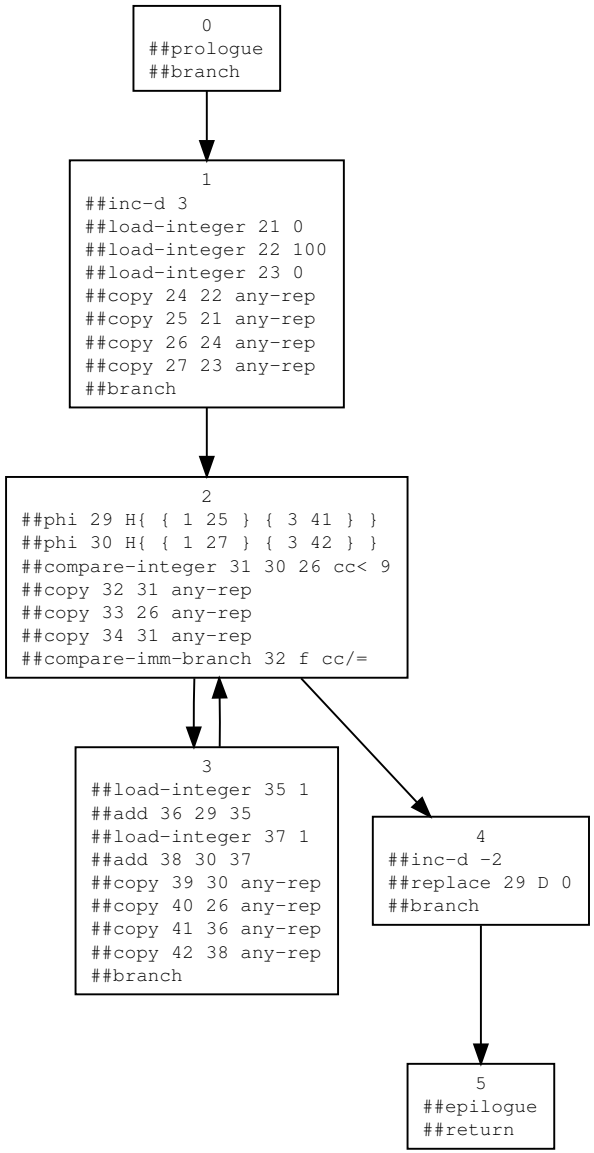
Example (In Factor)

```
0 100 [ 1 fixnum+fast ] times
```

Example (In Java)

```
int i = 0;
for (int j = 0; j < 100; j++) {
  i += 1;
}
```

Local Value Numbering



Local Value Numbering

Basic Block 1

```
vregs>vns = H{ }
exprs>vns = H{ }
```

```
##inc-d 3
##load-integer 21 0
##load-integer 22 100
##load-integer 23 0
##copy 24 22 any-rep
##copy 25 21 any-rep
##copy 26 24 any-rep
##copy 27 23 any-rep
##branch
```

(no-op)

Local Value Numbering

Basic Block 1

```
vregs>vns = H{ { 21 21 } }
exprs>vns = H{ {  0 21 } }
```

```
##inc-d 3
##load-integer 21 0
##load-integer 22 100
##load-integer 23 0
##copy 24 22 any-rep
##copy 25 21 any-rep
##copy 26 24 any-rep
##copy 27 23 any-rep
##branch
```

```
(no-op)
>expr = 0
```

Local Value Numbering

Basic Block 1

```
vregs>vns = H{ { 21 21 } {  22 22 } }
exprs>vns = H{ {  0 21 } { 100 22 } }
```

```
##inc-d 3
##load-integer 21 0
##load-integer 22 100
##load-integer 23 0
##copy 24 22 any-rep
##copy 25 21 any-rep
##copy 26 24 any-rep
##copy 27 23 any-rep
##branch
```

```
(no-op)
>expr = 0
>expr = 100
```

Local Value Numbering

Basic Block 1

```
vregs>vns = H{ { 21 21 } {  22 22 } { 23 21 } }
exprs>vns = H{ {  0 21 } { 100 22 } }
```

```
##inc-d 3
##load-integer 21 0
##load-integer 22 100
##load-integer 23 0
##copy 24 22 any-rep
##copy 25 21 any-rep
##copy 26 24 any-rep
##copy 27 23 any-rep
##branch
```

```
(no-op)
>expr = 0
>expr = 100
>expr = 0
```

Local Value Numbering

Basic Block 1

```
vregs>vns = H{ { 21 21 } {  22 22 } { 23 21 } { 24 22 } ... }
exprs>vns = H{ {  0 21 } { 100 22 } }
```

```
##inc-d 3
##load-integer 21 0
##load-integer 22 100
##copy 23 21 any-rep
##copy 24 22 any-rep
##copy 25 21 any-rep
##copy 26 24 any-rep
##copy 27 23 any-rep
##branch
```

```
(no-op)
>expr = 0
>expr = 100
>expr = 0
...
...
...
...
```

Local Value Numbering

Basic Block 2

```
vregs>vns = H{ }
exprs>vns = H{ }
```

```
##phi 29 H{ { 1 25 } { 3 41 } }
##phi 30 H{ { 1 27 } { 3 42 } }
##compare-integer 31 30 26 cc< 9
##copy 32 31 any-rep
##copy 33 26 any-rep
##copy 34 31 any-rep
##compare-imm-branch 32 f cc/=
```

```
(no-op)
(no-op)
```

Local Value Numbering

Basic Block 2

```
vregs>vns = H{ { 30 30 } { 26 26 } { 31 31 } }
exprs>vns = H{ { { ##compare-integer 30 26 cc< } 31 } }
```

```
##phi 29 H{ { 1 25 } { 3 41 } }
##phi 30 H{ { 1 27 } { 3 42 } }
##compare-integer 31 30 26 cc< 9
##copy 32 31 any-rep
##copy 33 26 any-rep
##copy 34 31 any-rep
##compare-imm-branch 32 f cc/=
```

```
(no-op)
(no-op)
>expr = ...
```

Local Value Numbering

Basic Block 2

```
vregs>vns = H{ { 30 30 } { 26 26 } { 31 31 } ... }
exprs>vns = H{ { { ##compare-integer 30 26 cc< } 31 } }
```

```
##phi 29 H{ { 1 25 } { 3 41 } }
##phi 30 H{ { 1 27 } { 3 42 } }
##compare-integer 31 30 26 cc< 9
##copy 32 31 any-rep
##copy 33 26 any-rep
##copy 34 31 any-rep
##compare-imm-branch 32 f cc/=
```

```
(no-op)
(no-op)
>expr = ...
...
...
...
```

Local Value Numbering

Basic Block 2

```
vregs>vns = H{ { 30 30 } { 26 26 } { 31 31 } ... }
exprs>vns = H{ { { ##compare-integer 30 26 cc< } 31 } }
```

```
##phi 29 H{ { 1 25 } { 3 41 } }
##phi 30 H{ { 1 27 } { 3 42 } }
##compare-integer 31 30 26 cc< 9
##copy 32 31 any-rep
##copy 33 26 any-rep
##copy 34 31 any-rep
##compare-integer-branch 30 26 cc<
```

```
(no-op)
(no-op)
>expr = ...
...
...
...
```

Local Value Numbering

Basic Block 3

```
vregs>vns = H{ }
exprs>vns = H{ }
```

```
##load-integer 35 1
##add 36 29 35
##load-integer 37 1
##add 38 30 37
##copy 39 30 any-rep
##copy 40 26 any-rep
##copy 41 36 any-rep
##copy 42 38 any-rep
##branch
```

Local Value Numbering

Basic Block 3

```
vregs>vns = H{ { 35 35 } }
exprs>vns = H{ {  1 35 } }
```

```
##load-integer 35 1
##add 36 29 35
##load-integer 37 1
##add 38 30 37
##copy 39 30 any-rep
##copy 40 26 any-rep
##copy 41 36 any-rep
##copy 42 38 any-rep
##branch
```

```
>expr = 1
```

Local Value Numbering

Basic Block 3

```
vregs>vns = H{ { 35 35 } { 29 29 } { 36 36 } }
exprs>vns = H{ {  1 35 } { { ##add-imm 29 1 } 36 } }
```

```
##load-integer 35 1
##add 36 29 35
##load-integer 37 1
##add 38 30 37
##copy 39 30 any-rep
##copy 40 26 any-rep
##copy 41 36 any-rep
##copy 42 38 any-rep
##branch
```

```
>expr = 1
>expr = { ##add-imm 29 1 }
```

Local Value Numbering

Basic Block 3

```
vregs>vns = H{ { 35 35 } { 29 29 } { 36 36 }  { 37 35 } }
exprs>vns = H{ {  1 35 } { { ##add-imm 29 1 } 36 } }
```

```
##load-integer 35 1
##add-imm 36 29 1
##load-integer 37 1
##add 38 30 37
##copy 39 30 any-rep
##copy 40 26 any-rep
##copy 41 36 any-rep
##copy 42 38 any-rep
##branch
```

```
>expr = 1
>expr = { ##add-imm 29 1 }
>expr = 1
```

Local Value Numbering

Basic Block 3

```
vregs>vns = H{ { 35 35 } { 29 29 } { 36 36 }  { 37 35 } ... }
exprs>vns = H{ {  1 35 } { { ##add-imm 29 1 } 36 } ... }
```

```
##load-integer 35 1
##add-imm 36 29 1
##copy 37 35 any-rep
##add 38 30 37
##copy 39 30 any-rep
##copy 40 26 any-rep
##copy 41 36 any-rep
##copy 42 38 any-rep
##branch
```

```
>expr = 1
>expr = { ##add-imm 29 1 }
>expr = 1
>expr = { ##add-imm 30 1 }
```

Local Value Numbering

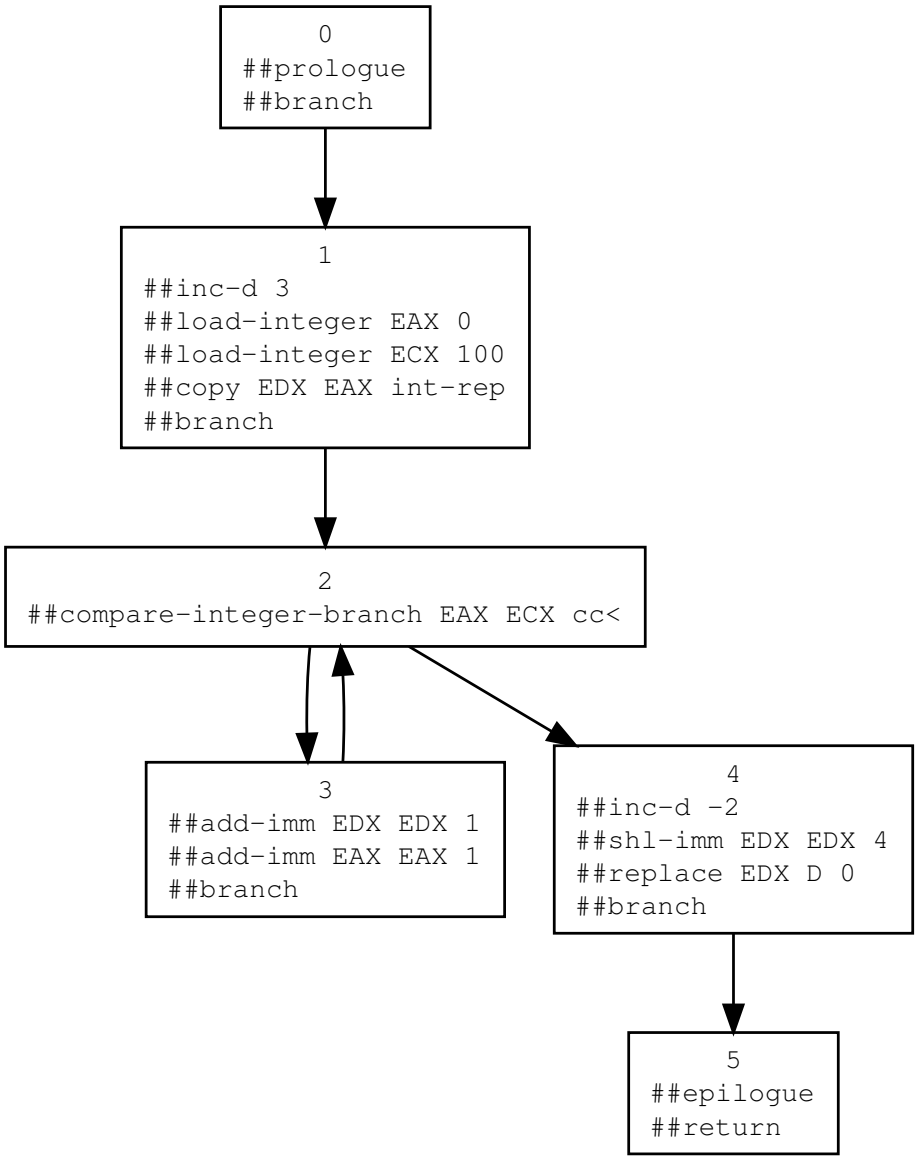
Basic Block 3

```
vregs>vns = H{ { 35 35 } { 29 29 } { 36 36 }  { 37 35 } ... }
exprs>vns = H{ {  1 35 } { { ##add-imm 29 1 } 36 } ... }
```

```
##load-integer 35 1
##add-imm 36 29 1
##copy 37 35 any-rep
##add-imm 38 30 1
##copy 39 30 any-rep
##copy 40 26 any-rep
##copy 41 36 any-rep
##copy 42 38 any-rep
##branch
```

```
>expr = 1
>expr = { ##add-imm 29 1 }
>expr = 1
>expr = { ##add-imm 30 1 }
...
...
...
...
```

Local Value Numbering Results



1. Compiler
   - Structure
   - Optimizations
2. Value Numbering
   - Local Value Numbering
   - Global Value Numbering
3. Results