VRP in LLVM

The PredicateSimplifier pass

by Nick Lewycky

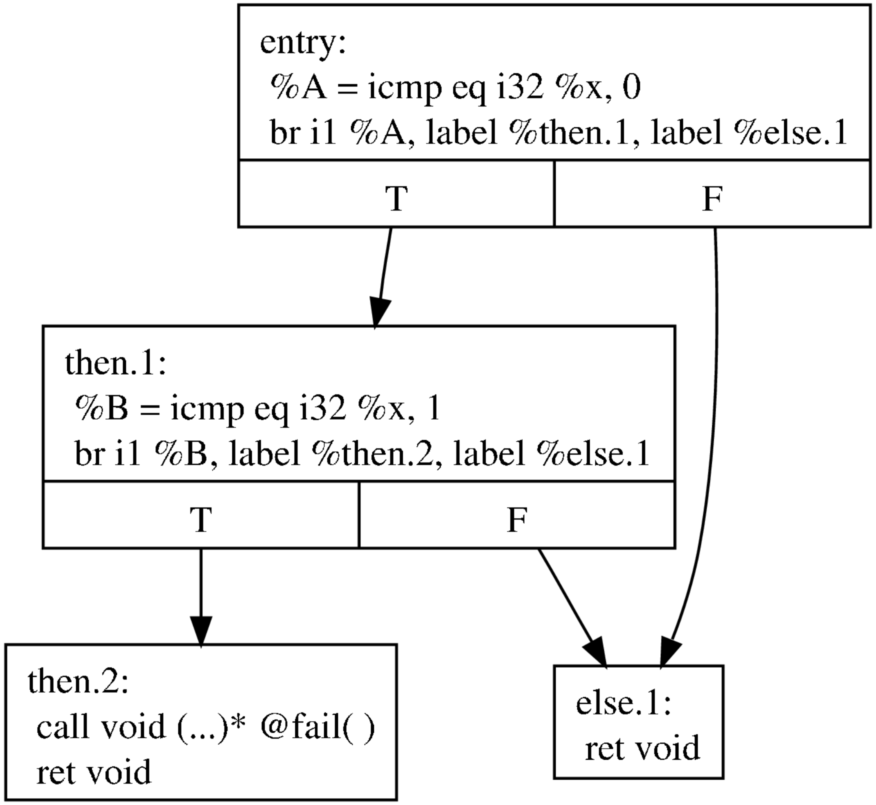
Motivation

```
void process(int *p, int *q) {
  if (p != q) return;

  if (*p != *q) f(); // not eliminated by LLVM
}
```

Demotivation

LLVM already optimizes some VRP examples!



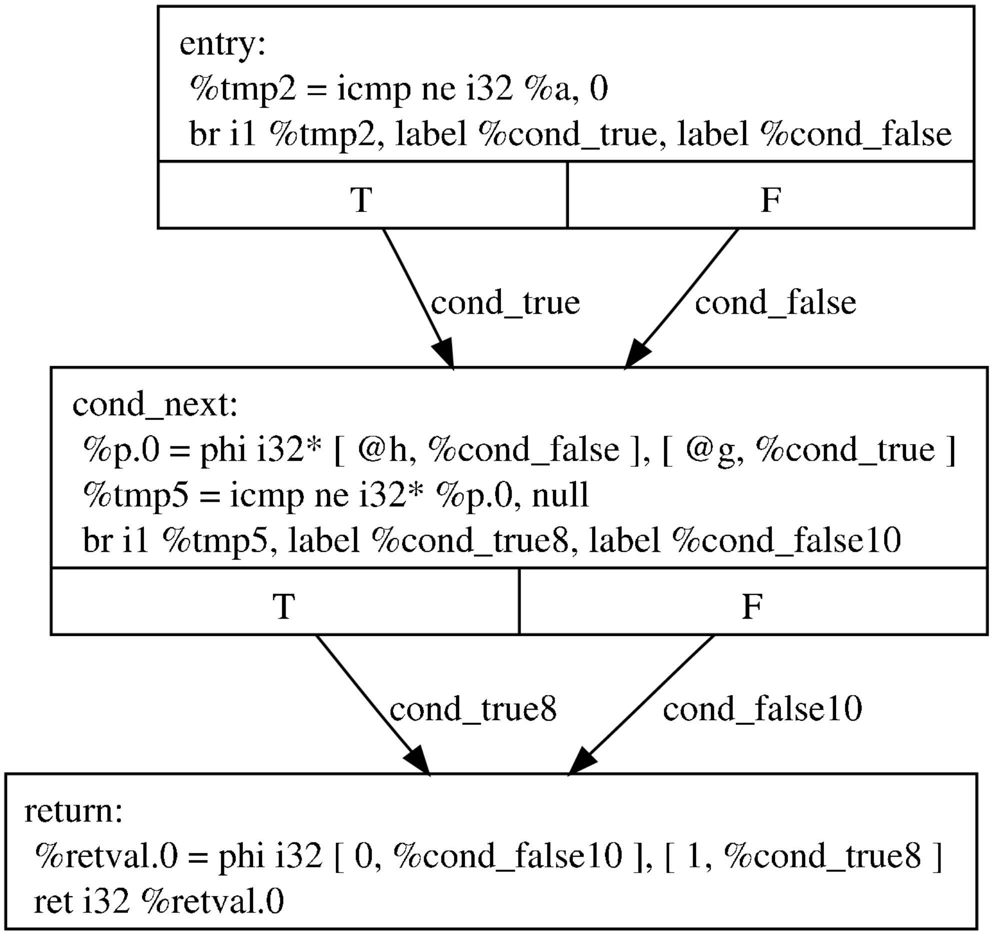
```
entry:
 %A = icmp eq i32 %x, 0
 br i1 %A, label %then.1, label %else.1
```

```
then.1:
 %B = icmp eq i32 %x, 1
 br i1 %B, label %then.2, label %else.1
```

```
then.2:
 call void (...)* @fail( )
 ret void
```

```
else.1:
 ret void
```

CFG for 'test1' function



```
entry:
 %tmp2 = icmp ne i32 %a, 0
 br i1 %tmp2, label %cond_true, label %cond_false
```

```
cond_next:
 %p.0 = phi i32* [ @h, %cond_false ], [ @g, %cond_true ]
 %tmp5 = icmp ne i32* %p.0, null
 br i1 %tmp5, label %cond_true8, label %cond_false10
```

```
return:
 %retval.0 = phi i32 [ 0, %cond_false10 ], [ 1, %cond_true8 ]
 ret i32 %retval.0
```

CFG for 'gcc21090' function

The PredicateSimplifier

The PredicateSimplifier is expected to grow into a full-fledged VRP pass some day. Currently, it does:

- Variable canonicalization
- Inequality graph
- Value ranges

Symbolic Execution

The pass begins at the entry block and proceeds, instruction by instruction, “executing” the code and maintaining properties.

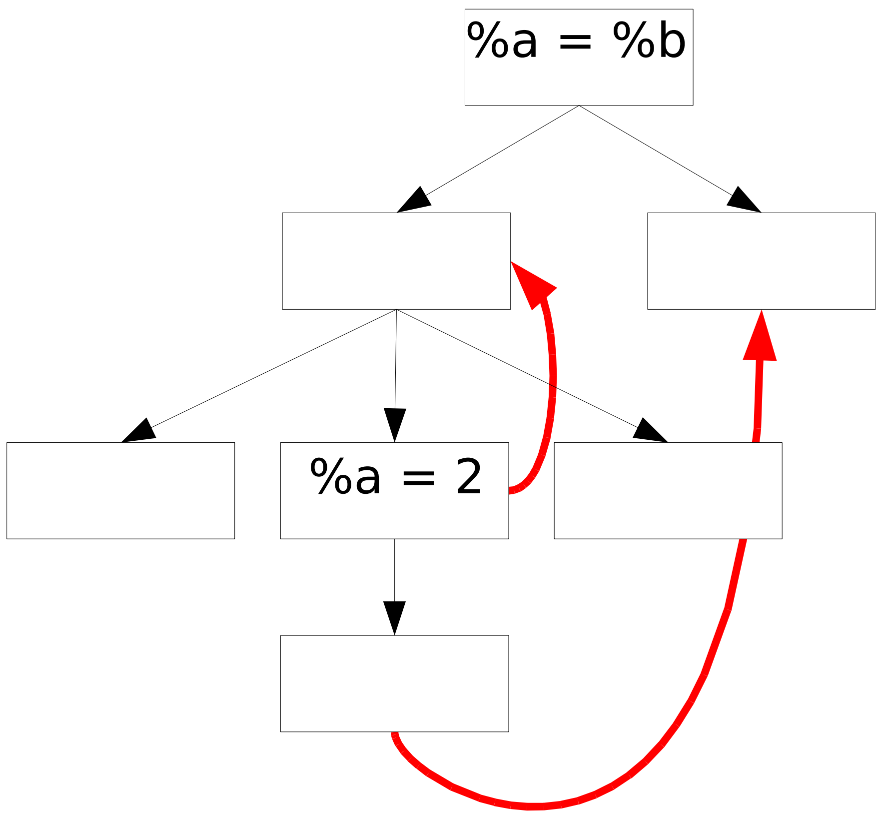
For example, “%a = udiv i8 %x, %y” implies that %y ≠ 0, and a branch on %P implies that %P is true in one side, false in the other.

Variable Canonicalization

The first thing that PredicateSimplifier does is canonicalize variables. Constants are best, then Arguments, then Instructions are compared by dominance.

If we know that %a = %b, then pick one and delete the other. All equal variables point to the same “Node” object storing the canonical choice.

Dominance



A value is only equal to another within a given scope.

We describe this scope as the set of dominated blocks.

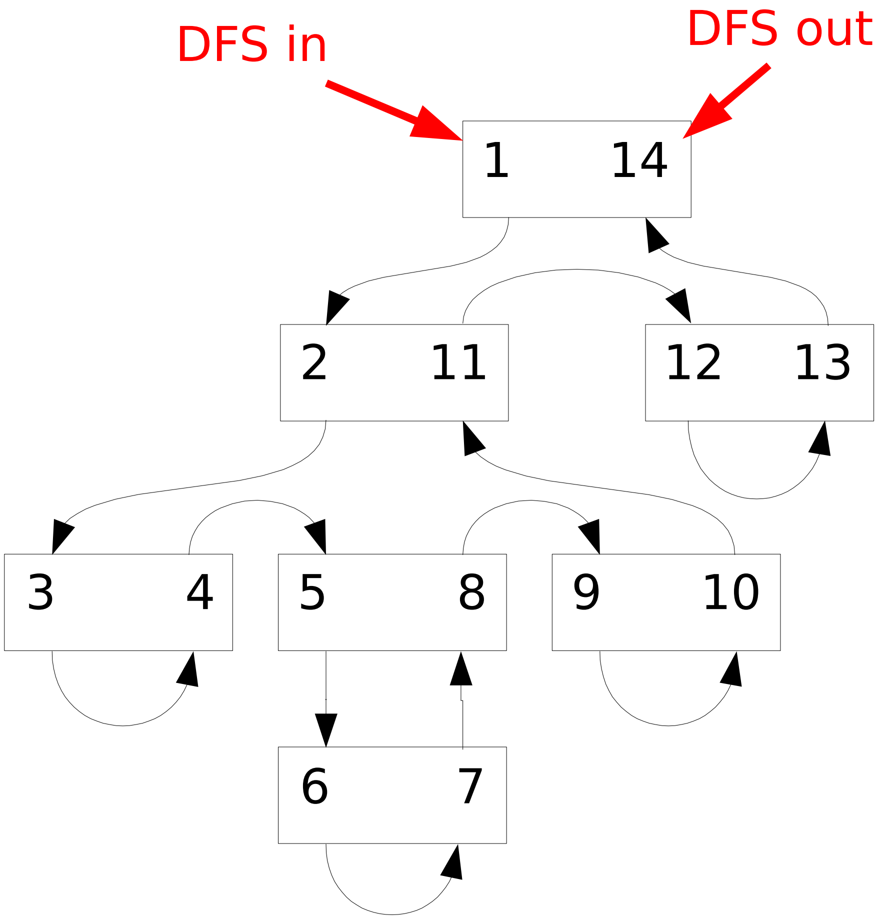
x dom y iff reaching *y* implies that control passed through *x*.

Depth First Numbering

Sort by “spread”, *DFS out – DFS in*.

A linear scan finds more specific (dominating fewer blocks) properties first.

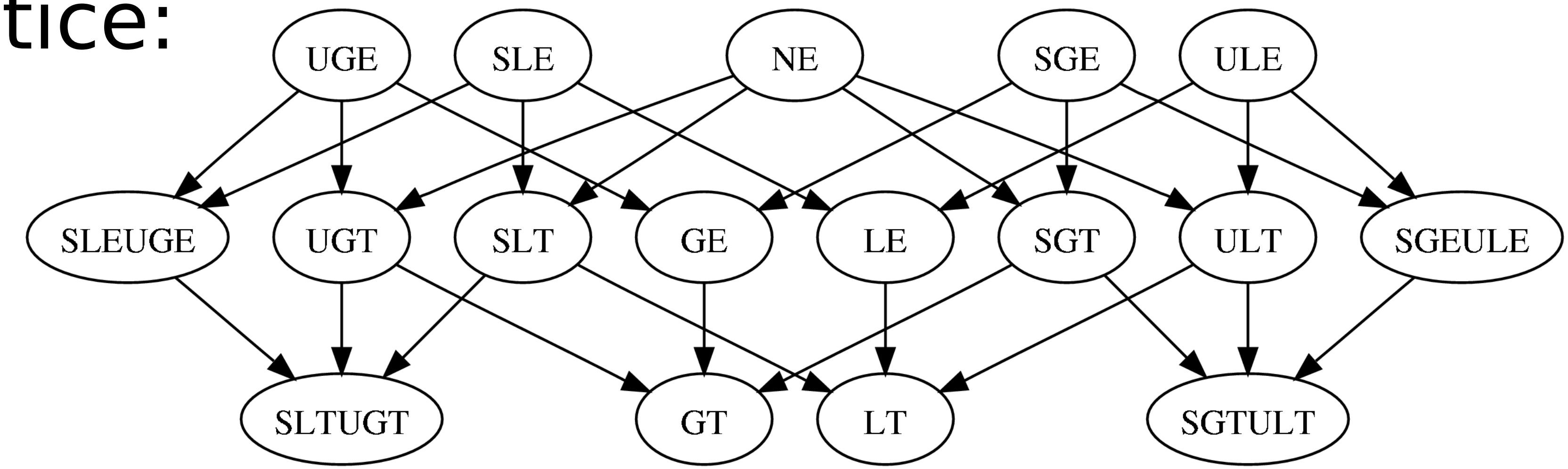
Canonical values are the ones with largest spread.



Inequality Graph (1/2)

Every canonical variable is a node in the inequality graph.

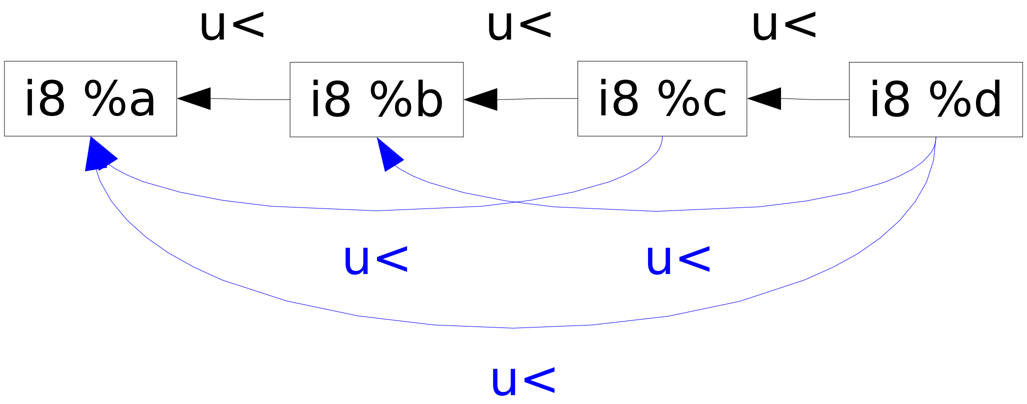
These nodes store their set of edges with the other nodes. The edges form a semi-lattice:



Inequality Graph (2/2)

The graph is stored with all transitive closures filled in (except != relationships). If %a u< %b u< %c, we also add %a u< %c.

Similarly, if %a u≤ %b u<s> %c then %a u< %c.



Value Ranges

Efficiently stores the range of possible values for a variable in terms of hard numbers.

Given i8 %a u< %b then %a in [0,254] and %b in [1,255].

These are stored with the same scoping technique as in the InequalityGraph.

The Work List (1/3)

The worklist stores the list of Instructions that need to be visited, as well as a bit of context information.

There is one “add” interface, used internally and externally to add new properties to this list. Its interface is modelled after the *icmp* instruction, and it takes a BB or Instruction for context.

The Work List (2/3)

There are two inspection methods that add new properties:

defToOps – Given that a new property has been found on an instruction definition, find new properties of the operands.

opsToDef – Given that a new property has been found on the operands, find a new properties on instruction.

The Work List (3/3)

The “context” is required to specify the scope in which the new properties will apply.

Consider “%a = or i1 %b, %c” in the entry block, then %a is found to be false under block %bb10. We can conclude that %b and %c are false only under %bb10.

Work-list Example

```
a = b + c
if (b == 5) { // reanalyzes “a”, finds nothing
    if (c == 3) {  // reanalyzes “a”, finds it
                   // equal to 8.
        use(a);   // this becomes use(8);
    }
    use(c);
}
use(b);
use(a);   // this stays use(a);
```

What's next?

- PHI nodes:
  - PHIs with operands dominate the PHI
  - PHIs that dominate their operands (loops)
- Improvements to the work list
- Floating-point number support
- Inter-procedural predsimplify

PHI Matrices

Given:

```
%a = phi(0 %bb1, %a.incr %bb2)
%b = phi(1 %bb1, 0 %bb2)
```

If we learn that %a = 10000 then we can conclude that %b = 0.

Expressions (1/2)

Consider:

```
%A = icmp ult %x, 5
%B = icmp ugt %x, 10
%bothcond = or i1 %A, %B
br i1 %A, label %cond_true, label %cond_false
```

Predsimplify will correctly determine that %cond_true is unreachable, but under %cond_false it won't show that %x can't be in [5, 10].

Expressions (2/2)

```
%bothcond = or i1 %A, %B
```

When visiting %cond_true, predsimplify assigns %bothcond to true, but then it stops. The result of an or statement being true tells you nothing about the operands.

The Trouble with FP

“Equals” in floating point is an inequality:

- “fcmp eq 0.0, -0.0” is true.
- “fcmp eq 0x7fc00000, 0x7fc00000” is false.

Just knowing that float %a eq float %b is not enough to perform variable canonicalization.

ip-predsimplify

Determining the range of possible returns of a call, relative to the arguments and global variables.

Proving BBs unreachable only useful if we can inline afterwards.

Should be very good at removing abstraction layers.