## Another example: integer range analysis

For each program point,
for each integer-typed variable,
calculate (an approximation to) the set of integer values that can be taken on by the variable

- use info for constant folding comparisons,
  for eliminating array bounds checks,
  for (in)dependence testing of array accesses,
  for eliminating overflow checks

What domain to use?

- what is its height?

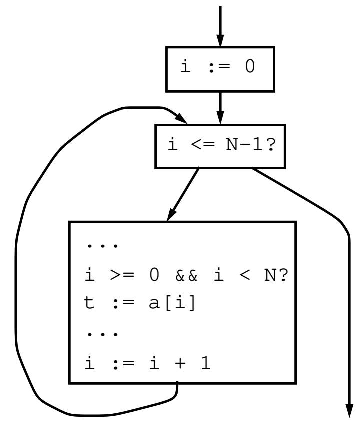
What flow functions to use?

- are they monotonic?



## Example

```
for i := 0 to N-1
  ... a[i] ...
end
```






## Widening operators

If domain is tall, then can introduce artificial generalizations (called **widenings**) when merging at loop heads

- can guarantee a small number of widenings possible



## Sharlit

A data flow analyzer generator [Tjiang & Hennessy 92]

- analogous to YACC

User writes basic primitives:

- control flow graph representation
  - nodes are instructions, not basic blocks
- domain (“flow value”) representation and key operations
  - `init`
  - `copy`
  - `is_equal`
  - `meet`
- flow functions for each kind of instruction
- action routines to optimize after analysis

Sharlit constructs iterative dataflow analyzer from these pieces

+ easy to build, extend
– not highly efficient

## Path compression

Can improve analysis efficiency by
summarizing effect of sequences of nodes

User can define path compression operations to collapse nodes together

- linear joining of sequential nodes
  ⇒ summarizes effect of whole BB
  - presumes a fixed GEN/KILL bit-vector structure to be effective
- merge trees into extended BB's
- merge merges, loops as in interval analysis
  - simplifies reducible parts, applies iteration to nonreducible parts

+ gets efficiency, preserves modularity & generality

– doesn't support data-dependent flow functions, cannot simulate optimizations during analysis

Performance results for code quality of generated optimizer, but not for compilation speed of optimizer



## Vortex IDFA framework

Like Sharlit, except a run-time library rather than a compiler

User defines a subclass of `AnalysisInfo` to represent elements of domain

- `copy`
- `merge` (lattice g.l.b. operator)
- `generalizing_merge` (g.l.b. with optional widening)
- `as_general_as` (lattice ≤ operator)

User invokes `traverse` to perform analysis:

```
cfg.traverse(direction, is_iterative?,
  initial_analysis_info,
  λ(rtl, info){ r.flow_fn(info) })
```

Flow function returns an `AnalysisResult`:

- keep instruction and continue analysis w/ updated info(s)
- delete instruction/constant-fold branch
- replace instruction with instruction or subgraph

`ComposedAnalysis` supports running multiple analyses interleaved at each instruction



## Features of Vortex IDFA

Big idea: separate analyses and transformations, make framework compose them appropriately

- don't have to simulate the effect of transformations during analysis
- can run analyses in parallel if each provides opportunities for the other
  - sometimes can achieve strictly better results this way than if run separately in a loop
- more general transformations supported (e.g. inlining) than Sharlit

Exploit inheritance & closures

Analysis speed is not stressed

- no path compression
- no "compilation" of analysis with framework