Classical Loop Optimizations

15-745 Optimizing Compilers
Spring 2006

Peter Lee

Reminders

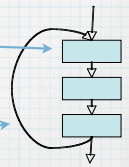
* Task 1 is due
* Task 2 available shortly
* Read 13.2 (loop-invariant code motion) and Ch.14 (induction variables)

Common loop optimizations

* **Hoisting of loop-invariant computations**
  * pre-compute before entering the loop
* **Elimination of induction variables**
  * change p=i*w+b to p=b+w, when w,b invariant
* **Elimination of null and array-bounds checks**
  * use laws of arithmetic to prove integer range
* **Loop unrolling**
  * to reduce number of control transfers
* **Loop permutation**
  * to improve cache memory performance

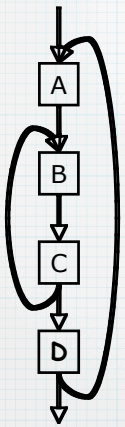
Finding loops

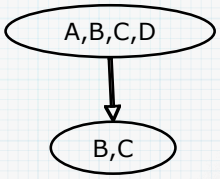
* To optimize loops, we need to find them!
* Specifically:
  * loop-header node(s)
    * nodes in a loop that have immediate predecessors not in the loop
  * back edge(s)
    * control-flow edges to previously executed nodes
  * all nodes in the loop body



Finding loops in L3

- L3 has only well-structured control-flow constructs
- Finding L3 loops is easy
  - the translator can mark every header node when creating the IR





In a **loop-nest tree**, each node represents the blocks of a loop, and parent nodes are enclosing loops

The leaves of the tree are the inner-most loops

Control-flow analysis

- Many languages have goto and other complex control, so loops can be hard to find in general
- Determining the control structure of a program is called **control-flow analysis**
- Based on the notion of **dominators**

Dominators

- **a dom b**
  - node **a** **dominates** **b** if every possible execution path from **entry** to **b** includes **a**
- **a sdom b**
  - **a** **strictly dominates** **b** if **a dom b** and **a≠b**
- **a idom b**
  - **a** **immediately dominates** **b** if there is no **c** such that **a sdom c** and **c sdom b**

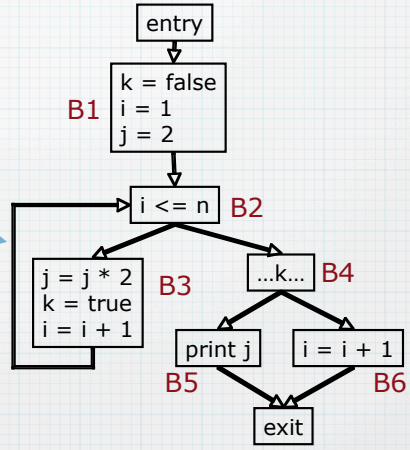
Some properties

- idom(n) is unique
- The dom relation is a partial ordering
  - reflexive, antisymmetric, and transitive

Back edges and loop headers

A control-flow edge from node B3 to B2 is a back edge if B2 dom B3

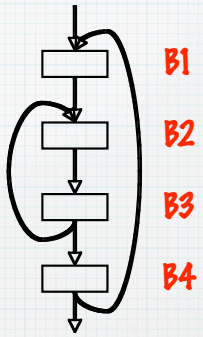
Furthermore, in that case node B2 is a loop header



Natural loop

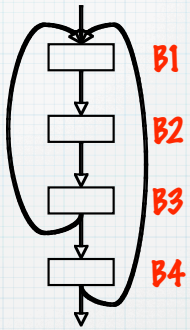
- Consider a back edge from node n to node h
- The natural loop of n→h is the set of nodes L such that for all x∈L:
  - h dom x and
  - there is a path from x to n not containing h

A simple example...



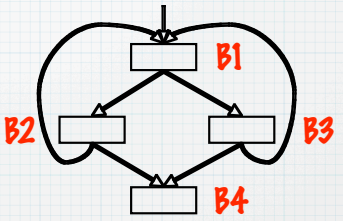
nested loops

What about this case?



loop with "continue"

What about this case?



conditional in loop

More later...

* We'll have more to say about dominators, including how to compute them efficiently, next week

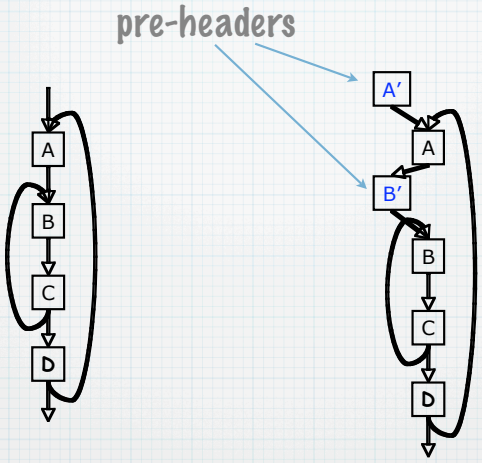
Loop optimizations:
Hoisting of loop-invariant computations

Loop-invariant computations

* A definition
  * t = x ⊕ y
* in a loop is (conservatively) loop-invariant if
  * x and y are constants, or
  * all reaching definitions of x and y are outside the loop, or
  * only one definition reaches x (or y), and that definition is loop-invariant

Hoisting

* In order to "hoist" a loop-invariant computation out of a loop, we need a place to put it
* We could copy it to all immediate predecessors (except along the back-edge) of the loop header...
* ...But we can avoid code duplication by inserting a new block, called the pre-header



Hoisting conditions

* For a loop-invariant definition
  * d: t = x ⊕ y
* we can hoist d into the loop's pre-header if
  1. d's block dominates all loop exits at which t is live-out, and
  2. there is only one definition of t in the loop, and
  3. t is not live-out of the pre-header

We need to be careful...

* All hoisting conditions must be satisfied!

```
L0:
  t = 0
L1:
  i = i + 1
  t = a * b
  M[i] = t
  if i<N goto L1
L2:
  x = t
```

OK

```
L0:
  t = 0
L1:
  if i>=N goto L2
  i = i + 1
  t = a * b
  M[i] = t
  goto L1
L2:
  x = t
```

violates 1,3

```
L0:
  t = 0
L1:
  i = i + 1
  t = a * b
  M[i] = t
  t = 0
  M[j] = t
  if i<N goto L1
L2:
```

violates 2

Loop optimizations:
Induction-variable elimination

The basic idea of IVE

* Suppose we have a loop variable
  * i initially 0; each iteration i = i + 1
* and a variable that linearly depends on it
  * $x = i * c_1 + c_2$
* In such cases, we can try to
  * initialize $x = i_0 * c_1 + c_2$
  * increment x by c_1 each iteration

Is it faster?

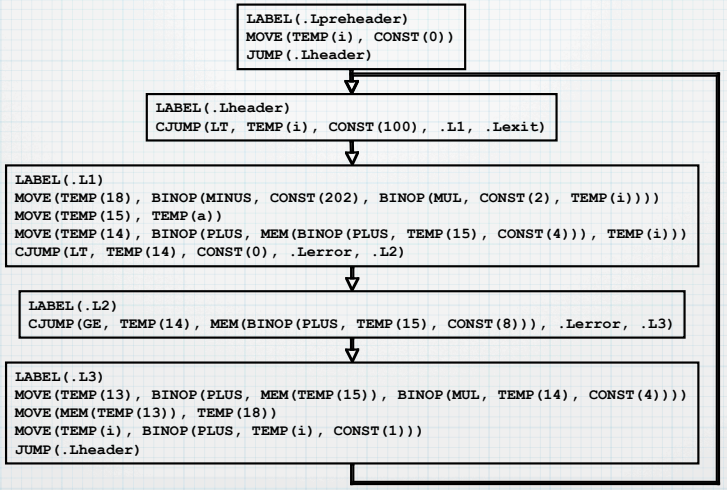
* On some hardware, adds are much faster than multiplies
* Furthermore, one fewer value is computed, thus potentially saving a register
  * and decreasing the possibility of spilling

An example

```
void p() {
  var a : int*;
  var i : int;

  a = alloc(100,int);
  for (i=0; i<100; i=i+1)
    a[i] = 202 - 2 * i;
}
```

IR for the loop



Loop preparation

* Before attempting IVE, it is best to perform first:
  * constant propagation & constant folding
  * copy propagation
  * loop-invariant hoisting

Copy propagation

```
y = x;
...
z = y + 1;
```

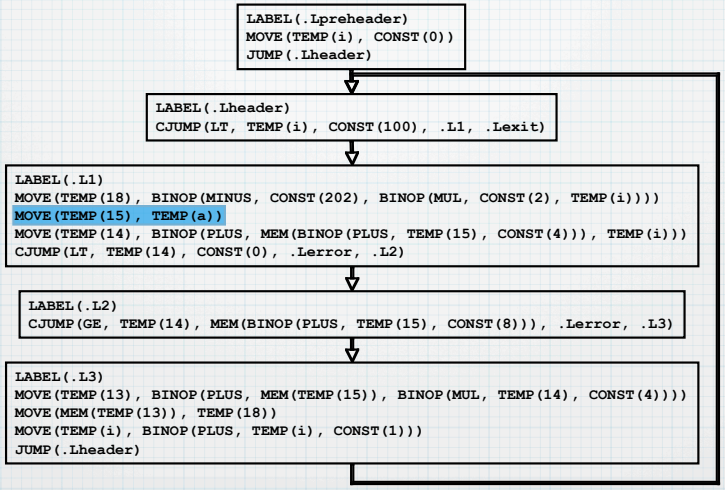
→

```
y = x;
...
z = x + 1;
```

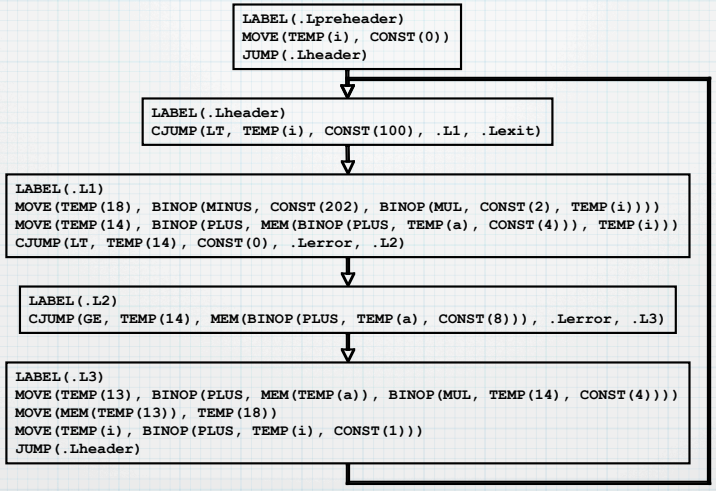
If there have been no changes to **x** after a copy into **y**, we can just use **x** instead of **y**

Dataflow analysis:
- forward analysis
- gen values: copy stmts of the form, **d:y=x**
- kill: if **x** is defined, then kill all **d** with **x** on rhs

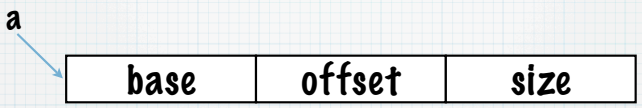
IR for the loop



After copy propagation



Fat pointer representation

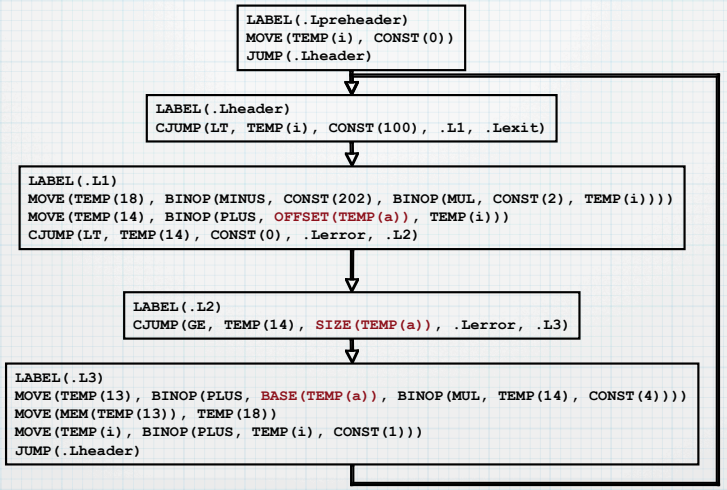


In typical implementations, the fields of a fat pointer are **immutable**

This means the base, offset, and size values for **a** can be considered loop-invariant

Sometimes convenient to reflect this in the IR...

With immutable fields



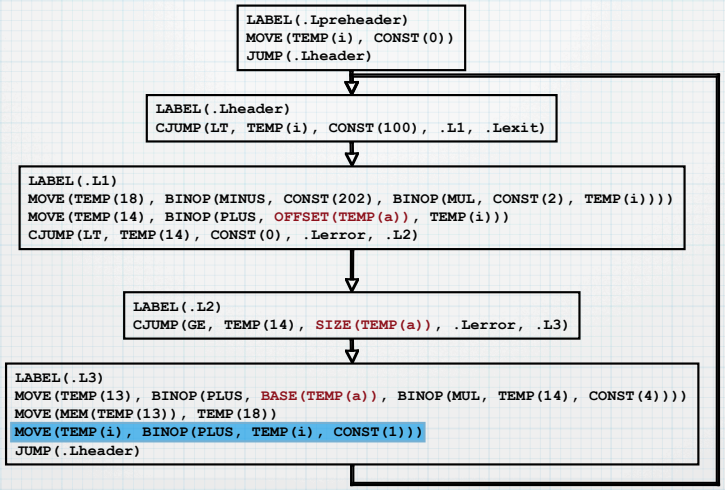
Induction variables

- Observe: Some variables change by a constant amount on each iteration
  - TEMP(i): init 0; increments by 1
  - TEMP(18): init 202; decrements by 2
  - TEMP(14): init offset(a); increments by 1
  - TEMP(13): init &a[offset(a)]; inc by 4
- These are all **induction variables**

Basic induction variables

- TEMP(i) is a **basic induction variable**
  - aka **independent induction variable**
- The only modifications to TEMP(i) are of the form:
  - TEMP(i) = TEMP(i) + c
  - where c is loop-invariant

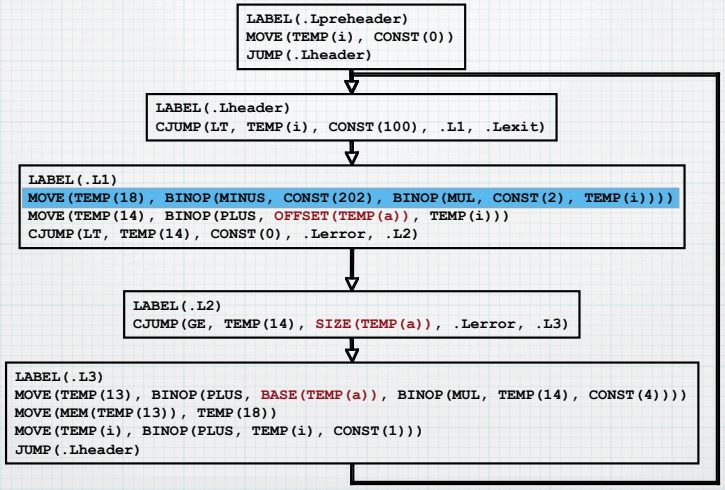
Basic induction variable



Derived induction variables

- TEMP(18) has the form
  - TEMP(18) = i * c_1 + c_2
- where
  - c_1, c_2 are loop-invariant, and
  - i is an induction variable
- TEMP(18) is a **derived induction variable**
  - aka **dependent induction variable**

Derived induction variable



The basic idea of IVE

* We have a basic IV
  * TEMP(i) = TEMP(i) + 1
* and derived IV
  * TEMP(18) = TEMP(i) * -2 + 202
* So, we can
  * initialize TEMP(18) = TEMP(i)_i * -2 + 202
  * increment TEMP(18) by 1*-2 each iteration

How to do it, step 1

* First, find the basic IVs
  * scan loop body for defs of the form
    * x = x + c, where c is loop-invariant
  * record these basic IVs as
    * x = (x, 1, c)
    * this represents the IV: x = x * 1 + c

How to do it, step 2

* Scan for derived IVs of the form
  * k = i * c_1 + c_2
  * where i is a basic IV and this is the only def of k in the loop
* We say **k is in the family of i**
* Record as k = (i, c_1, c_2)

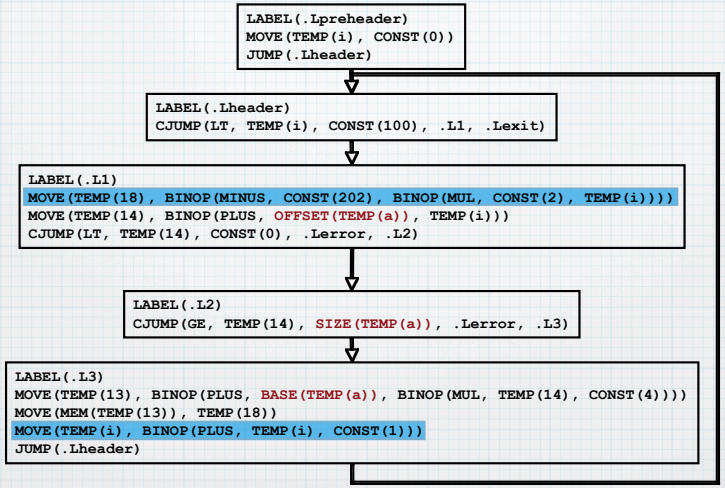
How to do it, step 3

- Iterate, looking for derived IVs of the form
  - $k = j * c_1 + c_2$
  - where IV $j = (i, a, b)$, and
  - this is the only def of k in the loop, and
  - there is no def of i between the def of j and the def of k
- Record as $k = (i, a*c_1, b*c_1+c_2)$

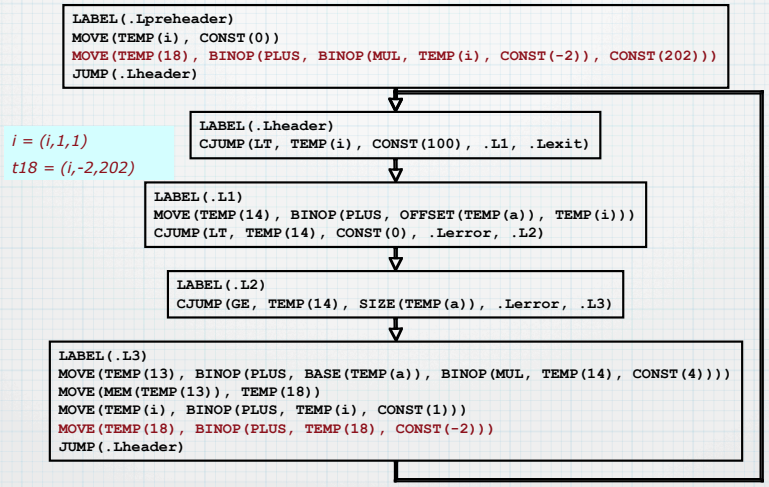
How to do it, step 4

- For an induction variable $k = (i, c_1, c_2)$
  - initialize $k = i * c_1 + c_2$ in the preheader
  - replace k's def in the loop by
    - $k = k + c_1$
  - make sure to do this after i's def

Back to our example

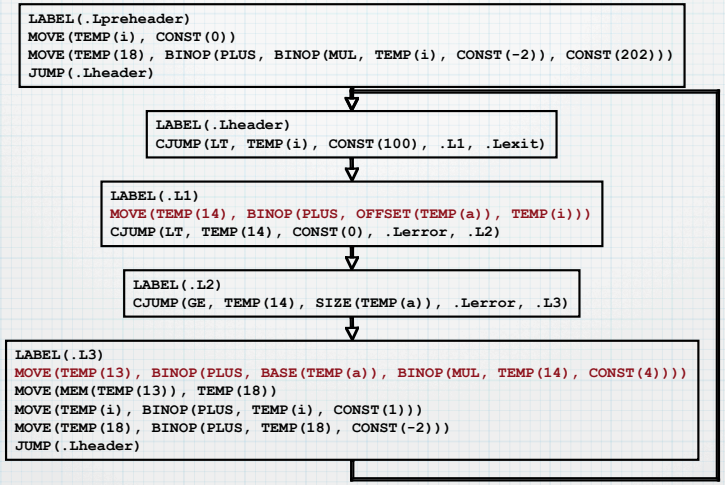


First round of IVE

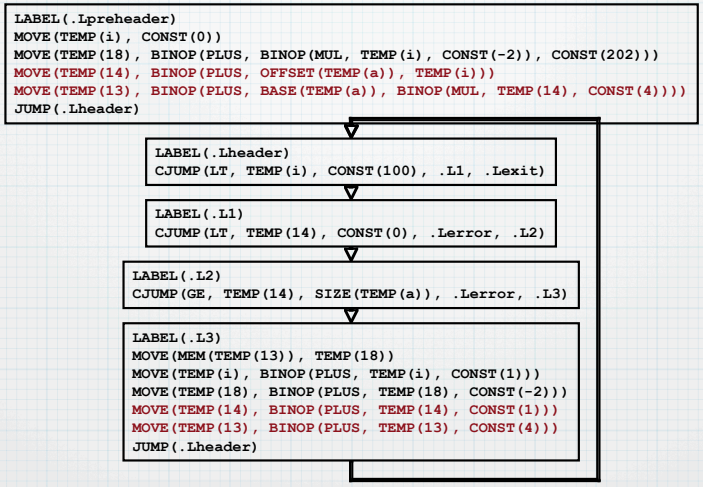


More IVE

i = (i,1,1)
t18 = (i,-2,202)
t14 = (i,1,offset(a))
*t13 = (i,4,offset(a)*4+base(a))*

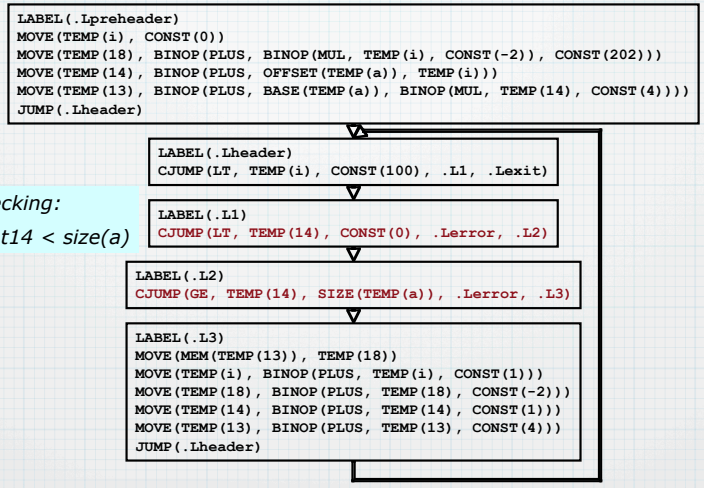


Final result



Loop optimizations:
Bounds-checking elimination

Bounds checks

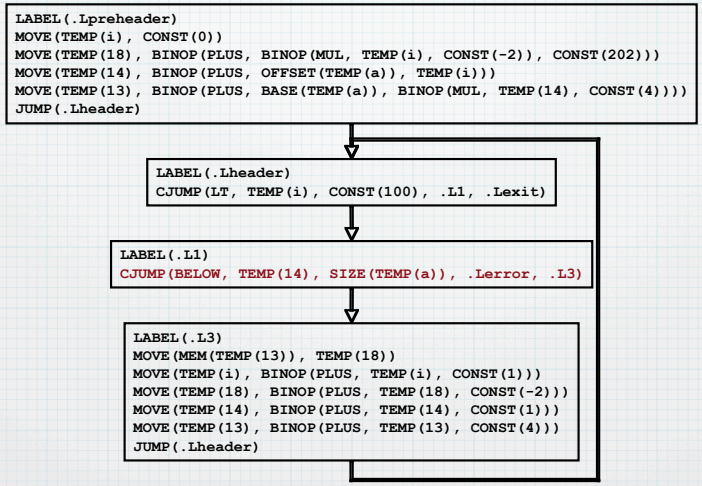


Checking:
$0 \le t14 < size(a)$

Bounds-check removal

* We want to verify that
  * 0 ≤ TEMP(14) < size(a)
* Observe: Since size(a) is definitely non-negative, we can implement the bounds check using a single unsigned comparison:
  * t14 $<_u$ size(1)

Using unsigned comparison



The simple case

* If index expression **i** is loop invariant, then we can hoist the bounds check into the preheader

Case: basic IV

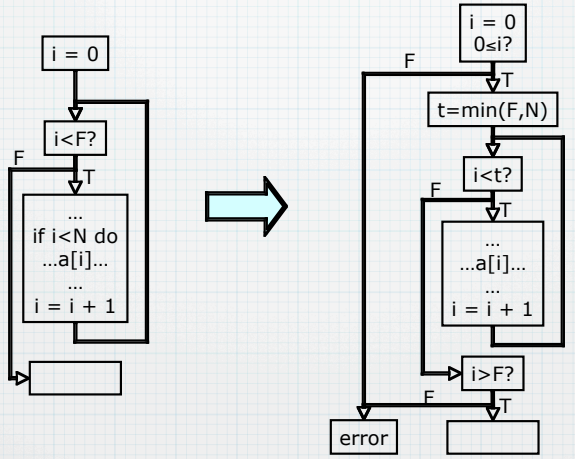
* If i is a basic IV, (i, 1, c), and i is used in a loop-test of the form
  * CJUMP(LT, i, F, L1, L2), or
  * CJUMP(GE, i, F, L2, L1), or
  * CJUMP(GT, F, i, L1, L2), or
  * CJUMP(LE, F, i, L2, L1)
    * where L2 is out of the loop
* Then we can...

loop-test bounds i

Basic IV case

- Then we can
  - insert a check for 0 ≤ i in the preheader
  - compute t = min(F, size(a))
    - either statically or in preheader
  - replace F with t in the loop test
  - add a check for i ≥ F at the loop exits

Typical situation



Our example

```
LABEL(.Lpreheader)
MOVE(TEMP(i), CONST(0))
MOVE(TEMP(18), BINOP(PLUS, BINOP(MUL, TEMP(i), CONST(-2)), CONST(202)))
MOVE(TEMP(14), BINOP(PLUS, OFFSET(TEMP(a)), TEMP(i)))
MOVE(TEMP(13), BINOP(PLUS, BASE(TEMP(a)), BINOP(MUL, TEMP(14), CONST(4))))
JUMP(.Lheader)
```

```
LABEL(.Lheader)
CJUMP(LT, TEMP(i), CONST(100), .L1, .Lexit)
```

```
LABEL(.L1)
CJUMP(BELOW, TEMP(14), SIZE(TEMP(a)), .Lerror, .L3)
```

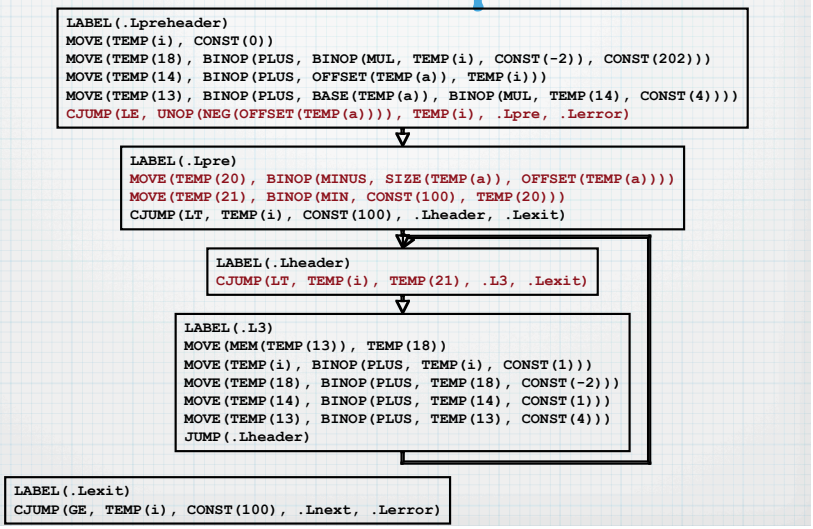
```
LABEL(.L3)
MOVE(MEM(TEMP(13)), TEMP(18))
MOVE(TEMP(i), BINOP(PLUS, TEMP(i), CONST(1)))
MOVE(TEMP(18), BINOP(PLUS, TEMP(18), CONST(-2)))
MOVE(TEMP(14), BINOP(PLUS, TEMP(14), CONST(1)))
MOVE(TEMP(13), BINOP(PLUS, TEMP(13), CONST(4)))
JUMP(.Lheader)
```

Case: derived IV

- If j is derived IV, (i, c_1, c_2), for i in the loop test
  - since $j = i*c_1 + c_2$, we can verify
    - $-c_2/c_1 \le i < (size(a) - c_2)/c_1$
- So:
  - insert check for $-c_2/c_1 \le i$ in preheader
  - compute $t = min(F, (size(a)-c_2)/c_1)$ in preheader
  - replace F with t in loop test
  - add a check for i>F in loop exits

Our example

i = (i,1,1)
t14 = (i,1,offset(a))



Other Approaches

Bounds checks are expensive

- Many, many approaches have been proposed to eliminate bounds checks
- The IVE-based method shown here is effective for loops whose indices are induction variables
- But many programs are not like this

Dataflow approaches

- We can try to use dataflow analysis
  - lattice elements: pairs (lo,hi)
  - $(lo1,hi1) \sqsubseteq (lo2,hi2)$ iff $lo1 \leq lo2$ and $hi1 \leq hi2$
  - $\bot$ is any (x,y) where x > y
  - $\top$ is $(-\infty, \infty)$
- This is the **range lattice**

Range lattice

* Unfortunately, the range lattice has infinite chains
* A typical ad hoc solution is to keep track of the number of updates to any variable during dataflow analysis and then "widen" to $(-\infty, \infty)$ after N iterations, for some N

Arithmetic-based approaches

* The dataflow approach is still ignorant of basic rules of arithmetic, however
* It is often profitable to analyze the program using a decision procedure for some fragment of arithmetic
* We will have some readings on this

Next week

* Partial redundancy elimination (PRE)
* Static single assignment form (SSA)