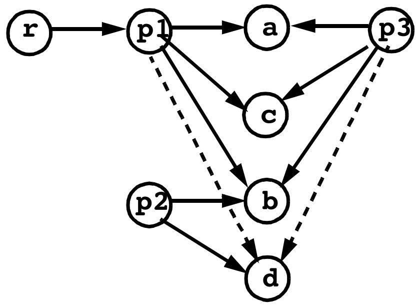
final assignment to `p2` can flow to `p1`, and then to `p3` through `r`:



This points-to graph is rather dense, but it does capture all the ways pointer values might propagate through the various pointer assignments.

Calls are handled by treating pointer parameters and pointer returns as assignments, done at the points of call and return.

Subprogram bodies are effectively inlined to capture the points-to relations they induce.

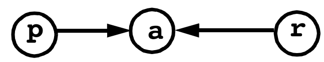
Given

```
*int echo (*int r) {
  return r; }
p = echo (&a);
```

we see the implicit assignments

```
r = &a;
p = r;
```

and add the following points-to information:



As an optimization, libraries can be pre-analyzed to determine the points-to relations they induce. Most may use (read) pointers but don't create any new points-to relations visible outside their bodies. Call to such library routines can be ignored as far as the caller's points-to graph is concerned.

Performance of Andersen's Algorithm

Experience has shown that Andersen's Algorithm gives useful points-to data and is far superior to the naive address-taken approach.

Interestingly, experiments show that making the technique flow-sensitive or calling context-sensitive doesn't improve results very much on typical benchmarks.

But execution time for moderate to large programs can be a problem.

Careful analysis shows that Andersen's Algorithm can require $O(n^3)$ time (where n is the number of nodes in the points-to graph).

The reason for this larger-than-expected analysis time is that a statement like

```
p = *q;
```

can force the algorithm to visit n^2 nodes (`q` may point to n nodes and each of these nodes may point to n nodes). The number of pointer statements analyzed can be O(n), leading to an $O(n^3)$ execution time.

Steensgaard's Algorithm

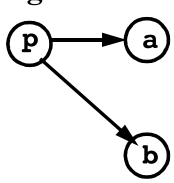
It would be useful to have a reasonably accurate points-to analysis that runs in essentially linear time so that really large programs could be handled.

This is what Steensgaard's Algorithm offers.

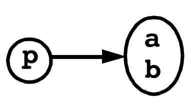
(Points-to Analysis in Almost Linear Time, B. Steensgaard, 1996 Principles of Programming Languages Conference.)

Steensgaard's Algorithm is essentially Andersen's Algorithm, simplified by merging nodes `a` and `b` if any pointer can reference both.

That is, in Andersen's Algorithm we might have

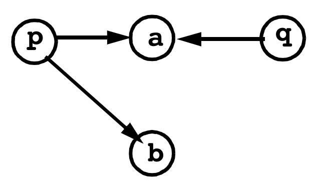


In Steensgaard's Algorithm we would instead have

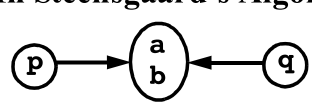


In effect any two locations that might be pointed to by the same pointer are placed in a single equivalence class.

Steensgaard's Algorithm is sometimes less accurate than Andersen's Algorithm. For example, the following points-to graph, created by Andersen's Algorithm, shows that `p` may point to `a` or `b` whereas `q` may only point to `a`:



In Steensgaard's Algorithm we get



incorrectly showing that if `p` may point to `a` or `b` then so may `q`.

But now statements like

```
p = *q;
```

can't force the algorithm to visit n^2 nodes, because multiple nodes referenced by the same pointer are always merged. Using the fast union-find algorithm, we can get an execution time of $O(n\ \alpha(n))$ which is essentially linear in n. Now very large programs can be analyzed, and without too much of a loss in precision.

Andersen vs. Steensgaard in Practice

- Horwitz and Shapiro examined 61 C programs, ranging in size from 300 to 24,300 lines.
- As expected, Steensgaard is less precise: On average points-to sets are 4 times bigger; at worst 15 times bigger.
- As expected, Andersen is slower. On average 1.5 times slower: at worst 31 times slower.
- Both are much better than the naive "address taken" approach.
- Bottom line: Use Andersen for small programs, use Steensgaard (or something else) for large programs.

Reading Assignment

- Read "Fast and Accurate Flow-Insensitive Points-To Analysis," by Shapiro and Horwitz.
(Linked from the class Web page.)

The Horwitz-Shapiro Approach

It would be nice to have a points-to analysis that is parameterizable, ranging between the accuracy of Andersen and the speed of Steensgaard.

Horwitz and Shapiro (Fast and Accurate Flow-Insensitive Points-To Analysis, 1997 Principles of Programming Languages Conference) present a technique intermediate to those proposed by Andersen and Steensgaard.

Horwitz and Shapiro suggest each node in the points-to graph be limited to out degree k, where $1 \le k \le n$.

If k =1 then they have Steensgaard's approach.

If k =n (n is number of nodes in points to graph), then they have Andersen's approach.

Their worst case run-time is

$O(k^2\ n)$, which is not much worse than Steensgaard if k is kept reasonably small.

To use their approach assign each variable that may be pointed to to one of k categories.

Now if `p` may point to `x` and `p` may also point to `y`, we merge `x` and `y` **only if** they both are in the same category.

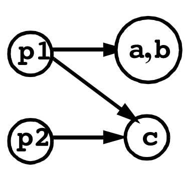
If `x` and `y` are in different categories, they **aren't** merged, leading to more accurate points-to estimates.

Example

```
p1 = &a;
p1 = &b;
p1 = &c;
p2 = &c;
```

Say we have k = 2 and place `a` and `b` in category 1 and `c` in category 2.

We then build:

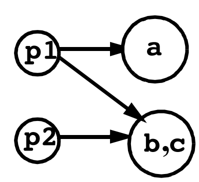


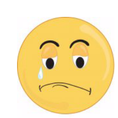
This points-to graph is just as accurate as that built by Andersen's approach.

But...

What if we chose to place `a` in category 1 and `b` and `c` in category 2.

We now have:





This graph is inexact, since it tells us `p2` may point to `b`, which is false.

(Steensgaard would have been worse still, incorrectly telling us `p2` may point to `a` as well as `b` and `c`).

Another Good Idea

What if we ran Shapiro and Horwitz's points-to analysis twice, each with different category assignments?

Each run may produce a different points-to graph. One may say `p2` points to `b` whereas the other says it does not.

Which do we believe?

Neither analysis misses a genuine points-to relation. Rather, merging of nodes sometimes creates false points-to information.

So we will believe `p2` may point to `b` only if all runs say so. This means multiple runs may "filter out" false points-to relations due to merging.

How Many Runs are Needed?

How are Categories to be Set?

We want to assign categories so that during at least one run, any pair of pointed-to variables are in different categories.

This guarantees that if all the runs tell us `p` may point to `a` and `b`, it is not just because `a` and `b` always happened to be assigned the same category.

To force different category assignments for each pair of variables, we assign each pointed-to variable an index and write that index in base k (the number of categories chosen).

For example, if we had variables `a`, `b`, `c` and `d`, and chose k = 2, we'd use the following binary indices:

`a` 00

`b` 01

`c` 10

`d` 11

Note that the number of base k digits needed to represent indices from 0 to n-1 is just ceiling($\log_k$ n).

This number is just the number of runs we need!

Why?

In the first run, we'll use the right most digit in a variable's index as its category.

In the next run, we'll use the second digit from the right, then the third digit from the right, ...

Any two distinct variables have different index values, so they must differ in at least digit position.

Returning to our example,

`a` 00

`b` 01

`c` 10

`d` 11

On run #1 we give `a` and `c` category 0 and `b` and `d` category 1.

On run #2, `a` and `b` get category 0 and `c` and `d` get category 1.

So using just 2 runs in this simple case, we eliminate much of the inaccuracy Steensgaard's merging introduces.

Run time is now $O(\log_k(n)\ k^2\ n)$.

How Well does this Approach Work?

On 25 tests, using 3 categories, Horwitz & Shapiro points-to sets on average are 2.67 larger than those of Andersen (Steensgaard's are 4.75 larger).

This approach is slower than Steensgaard but on larger programs it is 7 to 25 times faster than Andersen.

How Well do Points-to Analyses Work in Real Data Flow Problems?

In "Which Pointer Analysis Should I Use," Hind and Pioli survey the effectiveness of a number of points-to analyses in actual data flow analyses (mod/ref, liveness, reaching defs, interprocedural constant propagation).

Their conclusions are essentially the same across all these analyses:

- **Steensgaard's analysis is significantly more precise than address-taken analysis and not significantly slower.**
-

- **Andersen's analysis produces modest, but consistent, improvements over Steensgaard's analysis.**
- **Both context-sensitive points-to analysis and flow-sensitive points-to analysis give little improvement over Andersen's analysis.**

Reading Assignment

- Section 13.3 of *Crafting a Compiler*

"On the Fly" Local Register Allocation

Allocate registers as needed during code generation.

Partition registers into 3 classes.

- Allocatable

  Explicitly allocated and freed; used to hold a variable, literal or temporary.

  On SPARC: Local registers & unused In registers.

- Reserved

  Reserved for specific purposes by OS or software conventions.

  On SPARC: `%fp`, `%sp`, return address register, argument registers, return value register.

- Work

  Volatile—used in short code sequences that need to use a register.

  On SPARC: `%g1` to `%g4`, unused out registers.

Register Targeting

Allow "end user" of a value to state a register preference in AST or IR.

or

Use Peephole Optimization to eliminate unnecessary register moves.

or

Use *preferencing* in a graph coloring register allocator.

Register Tracking

Improve upon standard getReg/freeReg allocator by *tracking* (remembering) register contents.

Remember the value(s) currently held within a register; store information in a *Register Association List.*

Mark each value as *Saved* (in memory) or *Unsaved* (in memory).

Each value in a register has a *Cost*. This is the cost (in instructions) to restore the value to a register.

The cost of allocating a register is the sum of the costs of the values it holds.

$$\text{Cost(register)} = \sum_{\text{values} \in \text{register}} \text{cost(values)}$$

When we allocate a register, we will choose the *cheapest* one.

If 2 registers have the same cost, we choose that register whose values have the *most distant* next use.

(Why most distant?)

Costs for the SPARC

0	Dead Value
1	Saved Local Variable
1	Small Literal Value (13 bits)
2	Saved Global Variable
2	Large Literal Value (32 bits)
2	Unsaved Local Variable
4	Unsaved Global Variable

Register Tracking Allocator

```
reg getReg() {
  if ( ∃ r ∈ regSet and cost(r) == 0)
      choose(r)
  else {
      c = 1;
      while(true) {
         if ( ∃ r ∈ regSet and cost(r) == c){
             choose r with cost(r) == c and
                   most distant next use of
                   associated values;
             break;
          }
          c++;
      }
      Save contents of r as necessary;
  }
  return r;
}
```


- **Once a value becomes dead, it may be purged from the register association list without any saves.**
- **Values no longer used, but unsaved, can be purged (and saved) at *zero* cost.**
- **Assignments of a register to a simple variable may be *delayed*—just add the variable to the Register's Association List entry as unsaved.**

  The assignment may be done later or made *unnecessary* (by a later assignment to the variable)
- **At the end of a basic block all unsaved values are stored into memory.**

Example

```
int a,b,c,d; // Globals
a = 5;
b = a + d;
c = b - 7;
b = 10;
```

Naive Code

```
mov   5,%l0
st    %l0,[a]
ld    [a],%l0
ld    [d],%l1
add   %l0,%l1,%l1
st    %l1,[b]
ld    [b],%l1
sub   %l1,7,%l1
st    %l1,[c]
mov   10,%l1
st    %l1,[b]
```

18 instructions are needed (memory references take 2 instructions)

With Register Tracking

Instruction Generated	`%l0`	`%l1`
`mov 5,%l0`	5(S)	
`! Defer assignment to a`	5(S), a(U)	
`ld [d], %l1`	5(S), a(U)	d(S)
`!d unused after next inst`		
`add %l0,%l1,%l1`	5(S), a(U)	b(U)
`!b is dead after next inst`		
`sub %l1,7,%l1`	5(S), a(U)	c(U)
`! %l1 has lower cost`		
`st %l1, [c]`	5(S), a(U)	
`mov 10, %l1`	5(S), a(U)	b(U), 10(S)
`! save unsaved values`		
`st %l0, [a]`		b(U), 10(S)
`st %l1,[b]`		

12 instructions (rather than 18)

Pointers, Arrays and Reference Parameters

When an array, reference parameter or pointed-to variable is read, all unsaved register values that might be aliased must be *stored*.

When an array, reference parameter or pointed-to variable is written, all unsaved register values that might be aliased must be *stored*, then *cleared* from the register association list.

Thus if `a[3]` is in a register and `a[i]` is assigned to, `a[3]` must be stored (if unsaved) and removed from the association list.

Optimal Expression Tree Translation—Sethi-Ullman Algorithm

Reference: R. Sethi & J. D. Ullman, "The generation of optimal code for arithmetic expressions," Journal of the ACM, 1970.

Goal: Translate an expression tree using the *fewest* possible registers.

Approach: Mark each tree node, N, with an *Estimate* of the minimum number of registers needed to translate the tree rooted by N.

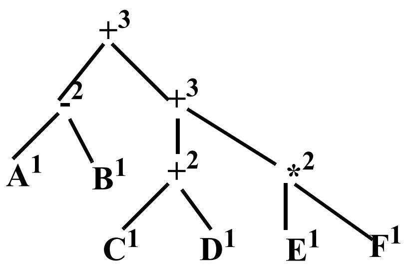
Let RN(N) denote the Register Needs of node N.

In a Load/Store architecture (ignoring immediate operands):

RN(leaf) = 1

RN(Op) =
If RN(Left) = RN(Right)
Then RN(Left) + 1
Else Max(RN(Left), RN(Right))

Example:



Key Insight of SU Algorithm

Translate subtree that needs more registers *first*.

Why?

After translating one subtree, we'll need a register to hold its value.

If we translate the more complex subtree first, we'll still have enough registers to translate the less complex expression (without *spilling* register values into memory).

Specification of SU Algorithm

TreeCG(tree *T, regList RL);

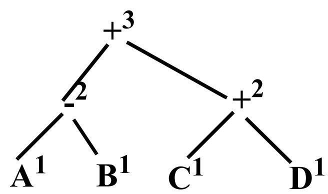
Operation:

- **Translate expression tree T using only registers in RL.**
- **RL must contain at least 2 registers.**
- **Result of T will be computed into head(RL).**

Summary of SU Algorithm

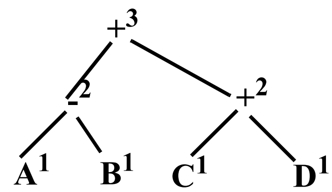
```
if T is a node (variable or literal)
    load T into R1 = head(RL)
else  (T is a binary operator)
  Let R1 = head(RL)
  Let R2 = second(RL)
  if  RN(T.left) >= Size(RL) and
      RN(T.right) >= Size(RL)
      (A spill is unavoidable)
       TreeCG(T.left, RL)
       Store R1 into a memory temp
       TreeCG(T.right, RL)
       Load memory temp into R2
       Generate (OP R2,R1,R1)
  elsif RN(T.left) >= RN(T.right)
       TreeCG(T.left, RL)
       TreeCG(T.right, tail(RL))
       Generate (OP R1,R2,R1)
  else
       TreeCG(T.right, RL)
       TreeCG(T.left, tail(RL))
       Generate (OP R2,R1,R1)
```

Example (with Spilling)



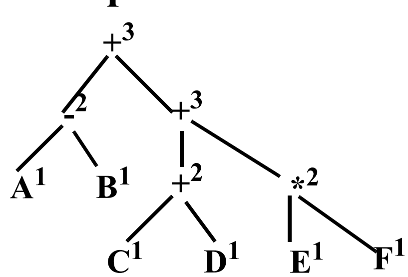
Assume only 2 Registers;
RL = [`%l0`,`%l1`]

We Translate the left subtree first (using 2 registers), store its result into memory, translate the right subtree, reload the left subtree's value, then do the final operation.



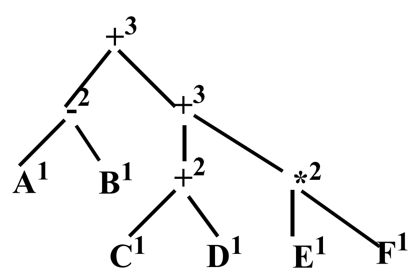

```
ld  [A], %l0
ld  [B], %l1
sub %l0,%l1,%l0
st  %l0, [temp]
ld  [C], %l0
ld  [D], %l1
add %l0,%l1,%l0
ld  [temp], %l1
add %l1,%l0,%l0
```


Larger Example



Assume 3 Registers;
RL = [`%l0`,`%l1`,`%l2`]

Since right subtree is more complex, it is translated first.




```
ld  [C], %l0
ld  [D], %l1
add %l0,%l1,%l0
ld  [E], %l1
ld  [F], %l2
mul %l1,%l2,%l1
add %l0,%l1,%l0
ld  [A], %l1
ld  [B], %l2
sub %l1,%l2,%l1
add %l1,%l0,%l0
```

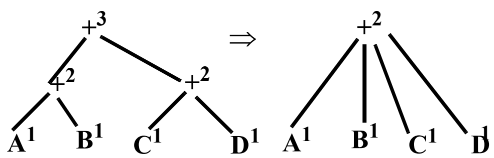
Refinements & Improvements

- **Register needs rules can be modified to model various architectural features.**

  For example, Immediate operands, that need not be loaded into registers, can be modeled by the following rule:

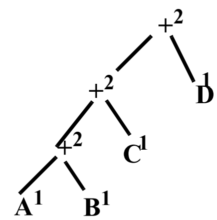
  RN(literal) = 0 if literal may be used as an immediate operand

- **Commutativity & Associativity of operands may be exploited:**



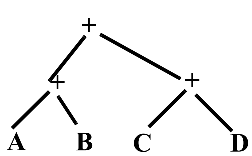
Is Minimizing Register Use Always Wise?

SU minimizes the *number* of registers used but at the *cost* of reduced ILP.



Since only 2 registers are used, there is little possibility of parallel evaluation.

When more registers are used, there is often more potential for parallel evaluation:



Here as many as *four* registers may be used to increase parallelism.

Optimal Translation for DAGs is Much Harder

If variables or expression values may be *shared* and *reused*, optimal code generation becomes NP-Complete.

Example: `a+b*(c+d)+a*(c+d)`

We must decide how long to hold each value in a register. Best orderings may "skip" between subexpressions

Reference: R. Sethi, "Complete Register Allocation Problems," *SIAM Journal of Computing*, **1975.**