Generalizing Data-flow Analysis

Announcements

- Project 2 writeup is available
- Read Stephenson paper

Last Time

- Control-flow analysis

Today

- C-Breeze Introduction
- Other types of data-flow analysis
  - Reaching definitions, available expressions, reaching constants
- Abstracting data-flow analysis
  What's common among the different analyses?
- Introduce lattice-theoretic frameworks

The C-Breeze Compiler

Characteristics

- A C source-to-source translator
- Implemented in C++
- Makes heavy use of templates and the Standard Template Library

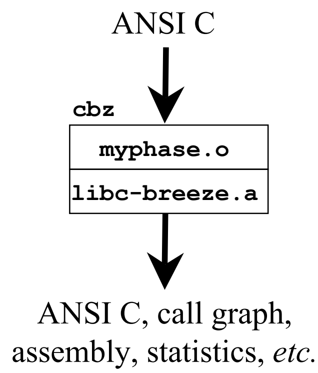
Organized as a set of phases

- Parse an ANSI C program, producing an AST
- Dismantle the C program into a *LIR* form
- Emit C code as output
- Can perform various analyses and transformations

Can create new phases easily

- Uses common design patterns: *walkers* and *changers*

C-Breeze Structure



Output is determined by the phases that are invoked

C-Breeze Terminology

Phase
- An organizational unit of work (analysis, transformation, or both)

Translation unit
- A file of a source program

Walker
- Uses visitor pattern to traversing the AST

Changer
- Uses visitor pattern for traversing and changing nodes in the AST

Walker Example (src/main/print_walker.*)

```
void print_walker::at_basicblock(basicblockNode * the_bb, Order ord)
{
  if (ord == Preorder) {
    indent(the_bb);
    _out << "basicblock: ";

    if (the_bb->decls().empty())
      _out << "(no decls) ";

    if (the_bb->stmts().empty())
      _out << "(no stmts) ";

    _out << endl;

    in();
  }

  if (ord == Postorder)
    out();
}
```

```
print_walker pw;
unit_list_p  u;

// for each translation unit in the program...
for (u=CBZ::Program.begin(); u!= CBZ::Program.end(); u++)
    // ... walk the AST using the CountAssg walker
    (*u)->walk (pw);
```


Other C-Breeze Information

No Symbol Table

Alias Analysis

```
int main()
{
    int a[10],b,c,d;

    a[3] = 4;
    b = 5+3;
    c = c+b + a[3];

    return 0;
}
```

```
...
 a[3] = 4;
    __T2 = 8;
    b = 8;
    __T3 = c + 8;
    __T4 = __T3 + a[3];
    c = __T4;
...
```

Evaluating Compiler Infrastructures

Efficiency
- Execution time for parsing and transformation
- Memory requirements
- Speed of resulting code

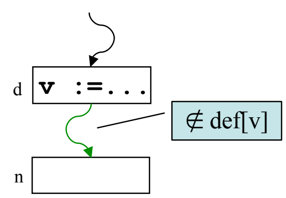
Front-ends and back-ends
- Which ones are available and how robust?

Intermediate Representation
- How well defined is the IR-API?

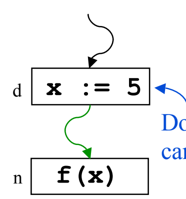
Pass Construction
- How difficult is it to write analyses and transformations?
- What basic analyses are available?
- Debugging assistance?

Reaching Definitions

Definition
- A definition (statement) d of a variable **`v`** **reaches** node n if there is a path from d to n such that **`v`** is not redefined along that path



Uses of reaching definitions
- Build use/def chains
- Constant propagation
- Loop invariant code motion



Does this def of **`x`** reach n?
can we replace n with **`f(5)`**?

```
1   a = . . .;
2   b = . . .;
3   for (. . .) {
4       x = a + b;
5       . . .
6   }
```

Reaching definitions of **`a`** and **`b`**

To determine whether it's legal to move statement 4 out of the loop, we need to ensure that there are no reaching definitions of **`a`** or **`b`** inside the loop

Computing Reaching Definitions

Assumption
- At most one definition per node
- We can refer to definitions by their node “number”

Gen[n]: Definitions that are generated by node n (at most one)
Kill[n]: Definitions that are killed by node n

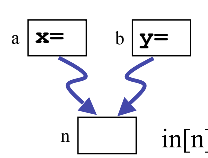
Defining Gen and Kill for various statement types

statement	Gen[s]	Kill[s]	statement	Gen[s]	Kill[s]
s: t = b op c	{s}	def[t] – {s}	s: goto L	{}	{}
s: t = M[b]	{s}	def[t] – {s}	s: L:	{}	{}
s: M[a] = b	{}	{}	s: f(a,…)	{}	{}
s: if a op b goto L	{}	{}	s: t=f(a, …)	{s}	def[t] – {s}

A Better Formulation of Reaching Definitions

Problem
- Reaching definitions gives you a set of definitions (nodes)
- Doesn’t tell you what variable is defined
- Expensive to find definitions of variable v

Solution
- Reformulate to include variable
  e.g., Use a set of (var, def) pairs

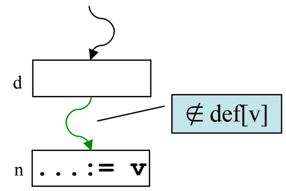


in[n] = {(x,a),(y,b),...}

Recall Liveness Analysis

Definition

- A variable is **live** at a particular point in the program if its value at that point will be used in the future (**dead**, otherwise).



Uses of Liveness

- Register allocation
- Dead-code elimination

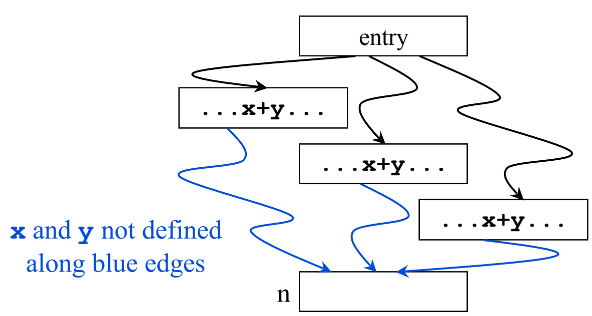
```
1   a = . . .;
2   b = . . .;
3   ...
4   x = f(b);
```

If **a** is not live out of statement 1 then statement 1 is dead code.

Available Expressions

Definition

- An expression, **x+y**, is **available** at node n if every path from the entry node to n evaluates **x+y**, and there are no definitions of **x** or **y** after the last evaluation



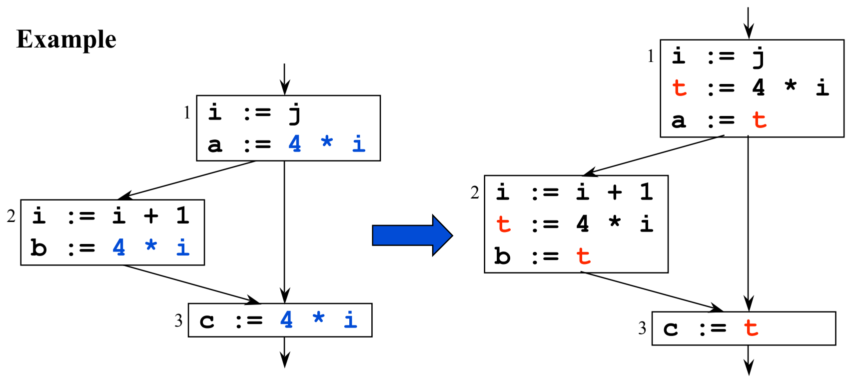
Available Expressions for CSE

How is this information useful?

Common Subexpression Elimination (CSE)

– If an expression is available at a point where it is evaluated, it need not be recomputed

Example



Aspects of Data-flow Analysis

Must or may Information	guaranteed or possible
Direction	forward or backward
Flow values	variables, definitions, ...
Initial guess	universal or empty set
Kill	due to semantics of stmt what is removed from set
Gen	due to semantics of stmt what is added to set
Merge	how sets from two control paths compose

Must vs. May Information

Must information

– Implies a guarantee

May information

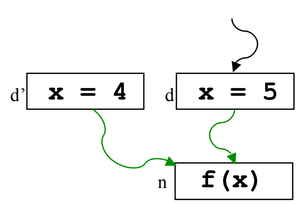
– Identifies possibilities

Liveness? Available expressions?

	May	Must
safe	overly large set	overly small set
desired information	small set	large set
Gen	add everything that might be true	add only facts that are guaranteed to be true
Kill	remove only facts that are guaranteed to be true	remove everything that might be false
merge	union	intersection
initial guess	empty set	universal set

Reaching Definitions: Must or May Analysis?

Consider constant propagation



We need to know if d' might reach node n

Defining Available Expressions Analysis

Must or may Information?	Must
Direction?	Forward
Flow values?	Sets of expressions
Initial guess?	Universal set
Kill?	Set of expressions killed by statement s
Gen?	Set of expressions evaluated by s
Merge?	Intersection

Reaching Constants

Goal
- Compute value of each variable at each program point (if possible)

Flow values
- Set of (variable,constant) pairs

Merge function
- Intersection

Data-flow equations
- Effect of node n **`x = c`**
  - kill[n] = {(x,d)| ∀d}
  - gen[n] = {(x,c)}
- Effect of node n **`x = y + z`**
  - kill[n] = {(x,c)| ∀c}
  - gen[n] = {(x,c) | c=valy+valz, (y, valy) ∈ in[n], (z, valz) ∈ in[n]}

Reality Check!

Some definitions and uses are ambiguous

- We can't tell whether or what variable is involved
  e.g., `*p = x;        /* what variable are we assigning?! */`
- Unambiguous assignments are called **strong updates**
- Ambiguous assignments are called **weak updates**

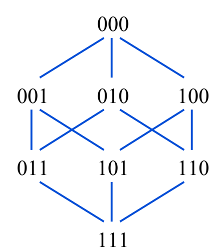
Solutions

- Be conservative
  - Sometimes we assume that it could be everything
    e.g., Defining **`*p`** (generating reaching definitions)
  - Sometimes we assume that it is nothing
    e.g., Defining **`*p`** (killing reaching definitions)
- Try to figure it out: alias/pointer analysis (more later)

Lattices

Define lattice $L = (V, \sqcap)$

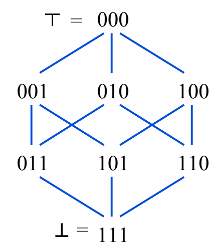
- V is a set of elements of the lattice
- $\sqcap$ is a binary relation over the elements of V (**meet** or **greatest lower bound**)



Properties of $\sqcap$

- $x,y \in V \Rightarrow x \sqcap y \in V$ (closure)
- $x,y \in V \Rightarrow x \sqcap y = y \sqcap x$ (commutativity)
- $x,y,z \in V \Rightarrow (x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$ (associativity)

Lattices (cont)



Under ($\sqsubseteq$)

- Imposes a partial order on V
- $x \sqsubseteq y \Leftrightarrow x \sqcap y = x$

Top ($\top$)

- A unique "greatest" element of V (if it exists)
- $\forall x \in V - \{\top\}, x \sqsubseteq \top$

Bottom ($\bot$)

- A unique "least" element of V (if it exists)
- $\forall x \in V - \{\bot\}, \bot \sqsubseteq x$

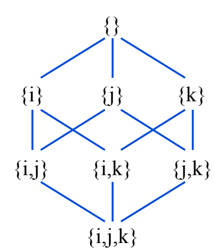
Height of lattice L

- The longest path through the partial order from greatest to least element (top to bottom)

Data-Flow Analysis via Lattices

Relationship

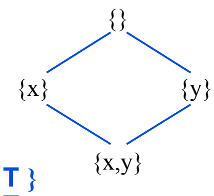
- Elements of the lattice (V) represent flow values (in[] and out[] sets)
  - *e.g.*, Sets of live variables for liveness
- $\top$ represents "best-case" information (initial flow value)
  - *e.g.*, Empty set
- $\bot$ represents "worst-case" information
  - *e.g.*, Universal set
- $\sqcap$ (meet) merges flow values
  - *e.g.*, Set union
- If $x \sqsubseteq y$, then x is a conservative approximation of y
  - *e.g.*, Superset



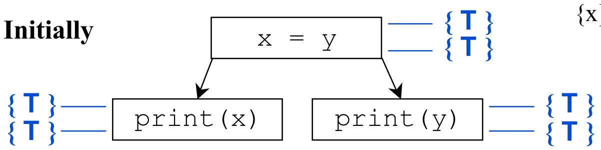
Data-Flow Analysis via Lattices (cont)

Remember what these flow values represent

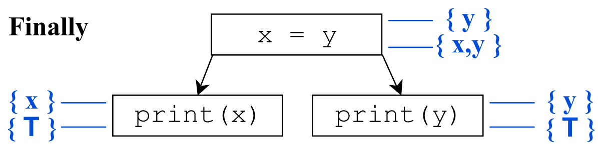
– At each program point a lattice element represents an in[] set or an out[] set



Initially



Finally



Data-Flow Analysis Frameworks

Data-flow analysis framework

– A set of **flow values** (V)

– A binary **meet operator** ($\sqcap$)

– A set of **flow functions** (F) (also known as **transfer functions**)

Flow Functions

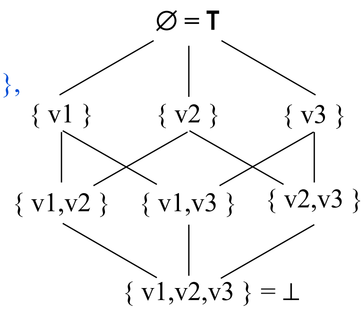
– F = {f: V→V}
  f describes how each node in CFG affects the flow values

– Flow functions map program behavior onto lattices

Visualizing DFA Frameworks as Lattices

Example: Liveness analysis with 3 variables
S = {v1, v2, v3}

- V: $2^S =$ {{v1,v2,v3}, {v1,v2},{v1,v3},{v2,v3}, {v1},{v2},{v3}, ∅}
- Meet (⊓): ∪
  - ⊑: ⊇
  - Top(T): ∅
  - Bottom (⊥): V
- F: $\{f_n(X) = Gen_n \cup (X - Kill_n), \forall n\}$



Inferior solutions are lower on the lattice
More conservative solutions are lower on the lattice

Concepts

Data-flow analyses are distinguished by
- Flow values (initial guess, type)
- May/must
- Direction
- Gen
- Kill
- Merge

Complication
- Ambiguous references (strong/weak updates)

Lattices
- Conservative approximation
- Optimistic (initial guess)
- Data-flow analysis frameworks
- Tuples of lattices