Last Class

- Worklist iterative algorithm for data-flow analysis
- Series of data-flow problems
  - LIVE, AVAIL, VERYBUSY, CONSTANTS
  - Example of non-distributive framework – CONSTANTS
  - Example of non-rapid framework – Interprocedural MOD

If we continue along this direction, the compiler solves one data-flow problem per transformation. Instead, we would like to solve one data-flow problem and use it for multiple transformations.

Instead, the community invented information chains
**Information Chains**

A tuple that connects 2 data-flow events is a *chain*

- Chains express data-flow relationships directly
- Chains provide a graphical representation
- Chains jump across unrelated code, simplifying search

We can build chains efficiently

Four interesting types of chain

<table>
<thead>
<tr>
<th>Source</th>
<th>Sink</th>
<th>Dependence Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>def</td>
<td>use</td>
<td>true, flow</td>
</tr>
<tr>
<td>use</td>
<td>def</td>
<td>anti</td>
</tr>
<tr>
<td>def</td>
<td>def</td>
<td>output</td>
</tr>
<tr>
<td>use</td>
<td>use</td>
<td>input</td>
</tr>
</tbody>
</table>

Def-Use chains are the most common

---

**Example**

```
 a ← 5
 b ← 3
 c ← b + 2
 d ← a - 2
 e ← a + b
 e ← e + c
 e ← 13

 f ← 2 + e
 Write f
```

d is dead
It has no use

def-use chains
**Notation**

Assume that, \( \forall \) operation \( i \) and each variable \( v \),

- \( \text{DEFS}(v,i) \) is the set of operations that may have defined \( v \) most recently before \( i \), along some path in the CFG
- \( \text{USES}(v,i) \) is the set of operations that may use the value of \( v \) computed at \( i \), along some path in the CFG

\[ x \in \text{DEFS}(A,y) \iff y \in \text{USES}(A,x) \]

To construct \( \text{DEF-USE} \) chains, we solve \textit{reaching definitions} (\textit{YADFP})

- A definition \( d \) of some variable \( v \) \textit{reaches} an operation \( i \) if and only if \( i \) reads \( v \) and there is a \( v \)-clear path from \( d \) to \( i \)
  - \( v \)-clear \( \Rightarrow \) no definition of \( v \) on the path
- Prior definition of \( v \) in same block \( \Rightarrow |\text{DEFS}(v,i)| = 1 \)
- No prior definition \( \Rightarrow |\text{DEFS}(v,i)| \geq 1 \)

The chains are non-local in this case

**Reaching Definitions**

The equations

\[ \text{REACHES}(n) = \emptyset, \forall \ n \in \mathcal{N} \]

\[ \text{REACHES}(n) = \bigcup_{p \in \text{preds}(n)} \text{DEDEF}(p) \cup (\text{REACHES}(p) \cap \text{DEFKILL}(p)) \]

- \( \text{REACHES}(n) \) is the set of definitions that reach block \( n \)
- \( \text{DEDEF}(n) \) is the set of definitions in \( n \) that reach the end of \( n \)
- \( \text{DEFKILL}(n) \) is the set of defs obscured by a new def in \( n \)

Computing \( \text{REACHES}(n) \)

- Use any data-flow method (\textit{rapid framework})
- Zadeck shows a simple linear-time scheme

\[ \text{F.K. Zadeck, “Incremental data-flow analysis in a structured program editor,”} \textit{Proceedings of the SIGPLAN 84 Conf. on Compiler Construction}, \text{June, 1984, pages 132-143.} \]
Building DEF sets

The Plan
1. Find basic blocks & build the CFG
2. \( \forall \) block \( b \), compute \( \text{REACHES}(b) \) (to the fixed point)
3. \( \forall \) block \( b \), \( \forall \) operation \( i \), \( \forall \) referenced name \( v \),
   Set \( \text{DEFS}(v,i) \) according to the earlier rule
   A.) If there is a prior definition, \( d \), of \( v \) in \( b \)
       \( \text{DEFS}(v,i) \leftarrow d \)
   B.) Otherwise
       \( \text{DEFS}(v,i) \leftarrow \{ d \mid d \text{ defines } v \text{ & } d \in \text{REACHES}(b) \} \)

To build USES
- Invert \( \text{DEFS} \), or
- Solve reachable uses, and use the analogous construction

Building DEF-USE Chains

Miscellany
- Domain of \( \text{REACHES} \) is the set of definitions \( \left( \propto |\text{operations}| \right) \)
- Domain of \( \text{DEFS} \& \text{USES} \) is also definitions
- Need a compact representation of \( \text{DEFS} \& \text{USES} \)

Detecting Anomalies
- \( \text{DEFS}(v,i) = \emptyset \) implies use of a never initialized variable
- Add a definition for each \( v \) to \( n_0 \) to detect larger set of anomalies
  - If initial def \( \in \text{DEFS}(v,i) \) then \( \exists \) a path to \( i \) with no initialization

And, how do we use these information chains?
Constant Propagation over DEF-USE Chains

Worklist ← ∅

while (Worklist ≠ ∅)
remove a definition i from WorkList
for each j ∈ USES(out, i)
set x so that out of i is in of j
Value(in₁,i) ← Value(in₁,j) ∧ Value(out,i)

if (Value(in₁,i) is a constant & Value(in₂,i) is a constant)
then Value(out,i) ← evaluate op i
Worklist ← Worklist ∪ { i }
else Value(out,i) ← T

Optimism

Initializations

Prior version used ⊥ (implicit)

In general

• Optimism helps inside loops
• Largely a matter of initial value

Constant Propagation over DEF-USE Chains

Complexity

• Initial step takes $O(1)$ time per operation
• Propagation takes
  > $|USES(v, i)|$ for each $i$ pulled from Worklist
  > Summing over all ops, becomes $|edges$ in DEF-USE graph$|$
  > A definition can be on the worklist twice (lattice height)
  > $O(|operations| + |edges$ in DU graph$|)$

This approach (on a sparse graph) is faster than the straightforward iterative approach in the Kildall style

Reminder of last lecture

Constant Propagation (Classic formulation)

Transformation: Global Constant Folding

• Along every path to $p$, $v$ has same known value
• Specialize computation at $p$ based on $v$'s value

Data-flow problem: Constant Propagation

Domain is the set of pairs $<v_i, c_i>$ where $v_i$ is a variable and $c_i \in C$

$$CONSTANTS(b) = \bigwedge_{p \in \text{preds}(b)} f_p(CONSTANTS(p))$$

• $\bigwedge$ performs a pairwise meet on two sets of pairs
• $f_p(x)$ is a block specific function that models the effects of block $p$ on the $<v_i, c_i>$ pairs in $x$

Constant propagation is a forward flow problem
**Example**

Meet operation requires more explanation

- \( c_1 \land c_2 = c_1 \) if \( c_1 = c_2 \), else \( \perp \) (bottom & top as expected)

What about \( f_p \) ?

- If \( p \) has one statement then
  
  \[
  x \gets y \text{ with } \text{CONSTANTS}(p) = \{\ldots<x,l_1>,\ldots<y,l_2>,\ldots\}
  \]
  
  then \( f_p(\text{CONSTANTS}(p)) = \text{CONSTANTS}(p) - <x,l_1> + <x,l_2> \)

- If \( p \) has \( n \) statements then

  \[
  f_p(\text{CONSTANTS}(p)) = f_n(f_{n-1}(f_{n-2}(\ldots f_2(f_1(\text{CONSTANTS}(p)))\ldots)))
  \]
  
  where \( f_i \) is the function generated by the \( i \)th statement in \( p \)

---

**Constant Propagation over DEF-USE Chains**

**Complexity**

- Initial step takes \( O(1) \) time per operation

- Propagation takes
  
  - \( |\text{USES}(v,i)| \) for each \( i \) pulled from Worklist
  
  - Summing over all ops, becomes \( |\text{edges in DEF-USE graph}| \)
  
  - A definition can be on the worklist twice (lattice height)
  
  - \( O(|\text{operations}| + |\text{edges in DU graph}|) \)

Can we do better?

- Not on the def-use chains …

- Would like to compute \( \land \) when new values are “born”
  
  - Where control flow brings chains together …
Constant Propagation over DEF-USE Chains

Birth points

We should be able to compute the values that we need with fewer meet operations, if only we can find these birth points.

- Need to identify birth points
- Need to insert some artifact to force the evaluation to follow the birth points
- Enter Static Single Assignment form

Constant Propagation over DEF-USE Chains

Making Birth Points Explicit

There are three birth points for x
**Constant Propagation over DEF-USE Chains**

Making Birth Points Explicit

Each needs a definition to reconcile the values of $x$

- Insert a $\phi$-function at each birth point
- Rename values so each name is defined once
- Now, each use refers to one definition

$\Rightarrow$ Static Single Assignment Form

---

**Constant Propagation over DEF-USE Chains**

Making Birth Points Explicit

How do we build SSA form?

- Simple algorithm
  1. Insert a $\phi$ at each join point for each name
  2. Rename to get single definition & single use

This produces

- Correct SSA form
- More $\phi$'s than any other known algorithm for SSA construction

The rest is optimization (!)

Next class:

SSA Construction

COMP 512, Spring 2009