Agenda

Quick recap (of everything so far):

- "Putting it all together" → Data-Flow Analysis
- Fixed-Point Iteration Strategies (3x)
- "Sign Analysis"
- "Constant Propagation Analysis"
- "Initialized Variables Analysis"
- Set-Based Analysis Framework
- WORKSHOP
DATA-FLOW ANALYSIS

- Undecidable problems (halting)
- Reduction principle
- Undecidability (R. Rice's Theorem)
- Static analysis (approximations)

MATH
- Crossproduct
- Relations
- Partial-orders
- Lattices
- Monotone functions
- Fixed-points (R. Fixed-point Theorem)

CONCEPTUAL MOTIVATION

CONTROL-SCIENCE
- Control-flow graphs
- Programming languages (syntax & semantics)

MAKE ANALYSES
- Faster programs!
- Find errors automatically!

PRACTICE!
- Appreciate & understand compilers & languages :-)

DATA-FLOW ANALYSIS

Aug 11, 2010
Claus Brabrand, UFPE, Brazil
Quick recap (of everything so far):

- "Putting it all together" → Data-Flow Analysis
- Fixed-Point Iteration Strategies (3x)
- "Sign Analysis"
- "Constant Propagation Analysis"
- "Initialized Variables Analysis"
- Set-Based Analysis Framework
- WORKSHOP

3 Example Data-Flow Analyses
All you need is...:

We (only) need 3 things:
- A control-flow graph
- A lattice
- Transfer functions

Given program:
```c
int x = 1;
int y = 3;
if (...) {
  x = x + 2;
} else {
  x <-> y;
}
print(x, y);
```

```
\[ \lambda E . E[x \mapsto 1] \]
\[ \lambda E . E[y \mapsto 3] \]
\[ \lambda E . E[x \mapsto E(x) \oplus 2] \]
\[ \lambda E . E[x \mapsto E(y), y \mapsto E(x)] \]
```

```
\[ \text{print}(x, y); \]
```

```
\[
\begin{array}{c}
\text{“top”} \\
-3 \\
-2 \\
-1 \\
0 \\
1 \\
2 \\
3 \\
\text{“bottom”} \\
\end{array}
\]
```

```
\[
\text{“could be anything”}
\]
```

```
\[
\text{“we haven’t analyzed yet”}
\]
```
\[\textbf{Solve Equations :-)\]}

- **One \textbf{big} lattice:**
  - E.g., \((L_{\text{VAR}})^{\text{PP}}\)

- **1 \textbf{big} abstract value vector:**
  \[
  \left[\begin{array}{c}
  \bot, \bot \\
  \bot, \bot \\
  \ldots \\
  \bot, \bot \\
  \end{array}\right] \in \left(\begin{array}{c}
  L_{\text{VAR}} \\
  \text{PP} \\
  \end{array}\right)
  \]

- **1 \textbf{big} transfer function:**
  - \(T : (L_{\text{VAR}})^{\text{PP}} \rightarrow (L_{\text{VAR}})^{\text{PP}}\)

- **Compute fixed-point (simply):**
  - Start with bottom value vector \((\bot, \bot, \ldots, \bot)\)
  - Iterate transfer function ‘\(T\)’ (until nothing changes)
  - Done; print out (or use) solution…! :-)

Claus Brabrand, UFPE, Brazil

DATA-FLOW ANALYSIS

The Entire Process :-) 

Program:

```plaintext
x = 0;
do {
    x = x+1;
} while (...);
output x;
```

1. Control-flow graph:

```plaintext
x = 0;
```

```
[Diagram of control-flow graph with nodes and edges]
```

3. Recursive equations:

```
a = \bot
b = f_{x=0}(a)
c = b \uplus d
d = f_{x=x+1}(c)
e = d
```

4. one "big" transfer function:

```
T((a,b,c,d,e)) = (\bot, f_{x=0}(a), b \uplus d, f_{x=x+1}(c), d)
```

5. Solve rec. equations...

```
T_0(\bot) T_1(\bot) T_2(\bot) T_3(\bot) T_4(\bot) = T_5(\bot)
```

```
[Diagram of power-lattice with nodes and edges]
```

```
|VAR| * |PP| = 1 * 5 = 5
```

...over a "big" power-lattice:

```
|VAR| * |PP| = 1 * 5 = 5
```

Another fixed point

Least fixed point
Exercise:

Repeat this process for program (of two vars):

```
x = 1;
y = 0;
while (v > w) {
    x <-> y;
}
y = y + 1;
```

i.e., determine…:

1) Control-flow graph
2) Transfer functions
3) Recursive equations
4) One ”big” transfer function
5) Solve recursive equations :-)

...using lattice:
Agenda

- Quick recap (of everything so far):
  - "Putting it all together" → Data-Flow Analysis
  - Fixed-Point Iteration Strategies (3x)
  - "Sign Analysis"
  - "Constant Propagation Analysis"
  - "Initialized Variables Analysis"
- Set-Based Analysis Framework
- WORKSHOP

3 Example Data-Flow Analyses
Naïve Fixed-Point Algorithm:

...uses intermediate "results" from previous iteration:

\[
\begin{align*}
\text{a} &= 0 \\
\text{b} &= \text{f}_{x=0}(\text{a}) \\
\text{c} &= \text{b} \cup \text{d} \\
\text{d} &= \text{f}_{x=x+1}(\text{c}) \\
\text{e} &= \text{d}
\end{align*}
\]

\[
\begin{align*}
\text{f}_{x=0}(L) &= 0 \\
\text{f}_{x=x+1}(L) &= L \oplus_L 0
\end{align*}
\]
Chaotic Iteration Algorithm

- Chaotic Iteration Algorithm:
  - ...exploits "forward nature" of program control flow:

\[
\begin{align*}
\text{Solution:} & \quad \Psi = \mathcal{L} = \mathcal{L} \oplus \mathcal{L} = \mathcal{L} \\
\text{Take:} & \quad \mathcal{L} = \mathcal{L} \\
\text{Compute:} & \quad \mathcal{L} = \mathcal{L} \oplus \mathcal{L} = \mathcal{L} \\
\text{Final:} & \quad \mathcal{L} = \mathcal{L} \oplus \mathcal{L} = \mathcal{L}
\end{align*}
\]

\[
\begin{align*}
a &= \bot \\
b &= f_{x=1}(a) \\
c &= b \cup d \\
d &= f_{x=x+1}(c) \\
e &= d
\end{align*}
\]

Faster!
(always uses "latest" results)

DATA-FLOW ANALYSIS
Work-list Algorithm

...uses a "queue" to control (optimize) computation:

Initialize queue with start point: 
\[ Q := [a] \]

Pop top element from queue and (re-)compute it; IF it changed THEN enqueue all points that depend on it's value (if it isn't already on the queue)

Stop when queue is empty

\[
\begin{align*}
    a &= \perp \\
    b &= f_{x=1}(a) \\
    c &= b \cup d \\
    d &= f_{x=x+1}(c) \\
    e &= d
\end{align*}
\]

Fastest! 
(in general)

\[
\begin{align*}
    f_{x=1}(\perp) &= \perp \\
    f_{x=x+1}(\perp) &= \perp \oplus \perp
\end{align*}
\]
Agenda

- Quick recap (of everything so far):
- "Putting it all together" → Data-Flow Analysis
- Fixed-Point Iteration Strategies (3x)
- "Sign Analysis"
- "Constant Propagation Analysis"
- "Initialized Variables Analysis"
- Set-Based Analysis Framework
- WORKSHOP

3 Example Data-Flow Analyses
The Language ‘C--’

- Syntactic Categories:

  - **Expressions** \((E \in EXP)\):
    
    \[
    E : n \mid v \mid E + E' \mid - E \\
    \mid E \ast E' \mid E == E' \mid \text{input}
    \]

  - **Statements** \((S \in STM)\):
    
    \[
    S : \text{skip} ; \mid v := E ; \mid \text{output E} ; \\
    \mid \text{if E then S else S'} \\
    \mid \text{while E do S} \mid \{ \text{var v; S}_1 \ldots \text{S}_n \}
    \]

- (…assume we only have integer variables ‘x’, ‘y’, ‘z’)

Claus Brabrand, UFPE, Brazil
Control-Flow Graph (for 'C--')

- Inductively defined control-flow graph:

  \[
  \begin{align*}
  \text{skip} & \quad; \\
  \text{v := E}_1 \quad; \\
  \text{output} \ E \quad; \\
  \text{if ( E ) S}_1 \text{ else S}_2 \\
  \text{while ( E ) S} \\
  \{ \text{var v; } S_1 \ldots S_n \}
  \end{align*}
  \]
Sign Analysis: Lattice

- Lattice:
  \[ \text{L}_{\text{SIGN}} \]

- \( \text{ENV}_{\text{Lattice}} \):
  \[ \begin{array}{ccc}
  & T & \\
  \cup & & \cup \\
  & 0 & \\
  \cup & & \cup \\
  \downarrow & & \downarrow \\
  x & & y & \times & \times & z
  \end{array} \]

- Confluence operator:
  \[ \sqcup = \left( \cup, \cup, \cup \right) \text{ (pairwise)} \]
|- Sign Analysis: Transfer F’s

- Transfer Functions:

\[ \lambda Env \cdot Env[x \mapsto T] \]

\[ \var x; \]

\[ \lambda Env \cdot Env \]

\[ \text{output ... ;} \]

\[ \lambda Env \cdot Env[x \mapsto \text{sign}(Env, Exp)] \]

\[ x := \text{Exp} ; \]
Inductive definition of 'sign' in the syntactic structure of Exp

Syntax:

\[
\begin{align*}
E : & \quad n \mid \text{input} \mid v \mid E + E' \\
& \quad E \star E' \mid E == E' \mid - E
\end{align*}
\]

- \(\text{sign}(\text{Env}, n) = \top\)
- \(\text{sign}(\text{Env}, \text{input}) = \bot\)
- \(\text{sign}(\text{Env}, v) = \text{Env}(v)\)
- \(\text{sign}(\text{Env}, E_1 + E_2) = \text{sign}(\text{Env}, E_1) \oplus_l \text{sign}(\text{Env}, E_2)\)
- \(\text{sign}(\text{Env}, -E) = \emptyset \Theta_l \text{sign}(\text{Env}, E)\)
- ...

...
Exercise:

- Come up with a program the analysis...

A)
- can analyse precisely

B)
- can’t analyse precisely
Agenda

- Quick recap (of everything so far):
  - "Putting it all together" → Data-Flow Analysis
- Fixed-Point Iteration Strategies (3x)
  - "Sign Analysis"
  - "Constant Propagation Analysis"
  - "Initialized Variables Analysis"
- Set-Based Analysis Framework
- WORKSHOP
Const Propagation: Lattice

- Lattice:
  \[ L_{\text{NUM}}: \]
  \[
  \begin{array}{c}
  \top \\
  \vdots \\
  \bot \\
  \end{array}
  \]

- \[ \text{ENV}_{L_{\text{NUM}}}: \]
  \[ \begin{array}{c}
  \begin{array}{c}
  \top \\
  \vdots \\
  \bot \\
  \end{array}
  \end{array} \times \begin{array}{c}
  \begin{array}{c}
  \top \\
  \vdots \\
  \bot \\
  \end{array}
  \end{array} \times \begin{array}{c}
  \begin{array}{c}
  \top \\
  \vdots \\
  \bot \\
  \end{array}
  \end{array} \]

- Confluence operator:
  - \[ \cup = \begin{array}{c}
    \begin{array}{c}
      \top \\
      \vdots \\
      \bot \\
    \end{array}
    \end{array} \begin{array}{c}
      \begin{array}{c}
      \top \\
      \vdots \\
      \bot \\
    \end{array}
    \end{array} \begin{array}{c}
      \begin{array}{c}
      \top \\
      \vdots \\
      \bot \\
    \end{array}
    \end{array} \] (pairwise)
Lt Const Propagation: Transfer F’s

Transfer Functions:

\[ \lambda Env . Env[x \mapsto \top] \]
\[ \text{var } x; \]

\[ \lambda Env . Env \]
\[ \text{output } ... ; \]

\[ \lambda Env . Env[x \mapsto \text{eval}(Env, Exp)] \]
\[ x := \text{Exp } ; \]
Inductive definition of \texttt{\textit{eval}} in the syntactic structure of \texttt{Exp}

- **Syntax:**

```
E : n | input | v | E + E' \\
| E * E' | E == E' | - E
```

- \texttt{eval(Env, n)} = n
- \texttt{eval(Env, input)} = \top
- \texttt{eval(Env, v)} = \texttt{Env(v)}
- \texttt{eval(Env, E_1+E_2)} = \texttt{eval(Env, E_1) \oplus_L eval(Env, E_2)}
- \texttt{eval(Env, -E)} = \emptyset \ominus_L \texttt{eval(Env, E)}
- ...

\[ n \oplus_L m = \begin{cases} 
\top, & \text{if } n = \top \lor m = \top \\
\bot, & \text{if } n = \bot \lor m = \bot \\
r, & \text{o/w (where } r = n + m) 
\end{cases} \]

...i.e.:
Exercise:

- Come up with a *program* the analysis...
  
  A)  
  - *can* analyse precisely

  B)  
  - *can’t* analyse precisely
- Agenda

- Quick recap (of everything so far):

- "Putting it all together" → **Data-Flow Analysis**

- Fixed-Point Iteration Strategies (3x)

- "Sign Analysis"

- "Constant Propagation Analysis"

- "Initialized Variables Analysis"

- Set-Based Analysis Framework

- WORKSHOP
- **Initialized Variables Analysis**

- **Lattice:**
  - Confluence operator: 
    - $\bigvee = \left( \bigvee, \bigvee, \bigvee \right)$ (pairwise)

- **ENV_{Lattice}**:

- **Confluence operator:**
  - $\bigvee = \left( \bigvee, \bigvee, \bigvee \right)$ (pairwise)
Initialized Variables Analysis

Transfer Functions:

\[ \lambda \text{Env} \cdot \text{Env}[x \mapsto \top] \]

\[ \text{var } x; \]

\[ \lambda \text{Env} \cdot \text{Env} \]

\[ \text{output } \ldots ; \]

\[ \lambda \text{Env} \cdot \text{Env}[x \mapsto \text{init}(\text{Env}, \text{Exp})] \]

\[ x := \ldots ; \]
Inductive definition of 'init' in the syntactic structure of Exp

Syntax:

\[
E : \begin{array}{llll}
  n & \text{input} & v & E + E' \\
  E \ast E' & E == E' & -E
\end{array}
\]

- \(init(Env, n) = \bot\)
- \(init(Env, \text{input}) = \bot\)
- \(init(Env, v) = Env(v)\)
- \(init(Env, E_1 + E_2) = eval(Env, E_1) \oplus_L eval(Env, E_2)\)
- \(init(Env, -E) = \bot \oplus_L eval(Env, E)\)
- ...
\[ \text{Note: Isomorphism!} \]

- With value lattice:

  \[
  \text{ENV-lattice is isomorphic to:} \quad \begin{array}{c}
  \text{uninitialized} = \top \\
  \text{initialized} = \bot
  \end{array}
  \]

- ENV-lattice is isomorphic to:

  \[
  \text{...for every program point:}
  \]

  \[
  \begin{array}{c}
  \text{vars that are possibly uninitialized}
  \end{array}
  \]

[Diagram of value lattice and ENV-lattice with annotations]
Initialized Variables Analysis (Revisited)

- ENV-lattice:

- Transfer Functions:
  \[ \lambda S \cdot S \cup \{x\} \]
  \[ \lambda S \cdot S \setminus \{x\} \]
  \[ \lambda S \cdot S \]

- Confluence operator:
  \[ \bigcup = \bigcup \text{ (i.e., set union)} \]

Vars that are possibly uninitialized
Exercise:

- Come up with a *program* the analysis...
  - A)
    - *can* analyse precisely
  - B)
    - *can’t* analyse precisely
Agenda

- Quick recap (of everything so far):
  - "Putting it all together" → Data-Flow Analysis
- Fixed-Point Iteration Strategies (3x)
  - "Sign Analysis"
  - "Constant Propagation Analysis"
- "Initialized Variables Analysis"
- Set-Based Analysis Framework
- WORKSHOP
Set-based analyses...

\{\text{may, must}\} \times \{\text{forwards, backwards}\}
Forwards vs. Backwards?

What you have seen:
- Forwards:
  - Some analyses…:

```
    x = 0;
    x = x+1;
    output x;
```

Analyze info that depends on past behavior

Some analyses…:
- Backwards:

```
    y = 2*x;
    x = x+1;
    output x;
```

Analyze info that depends on future behavior

E.g.:
- Live Variables
- Very Busy Expressions

'y' dead here
May vs. Must?

What you have seen:

- Confluence:
  - $\bigcup = \cup$ (set union)

- Partial order:
  - $\subseteq = \subseteq$ (sub-set-eq)

E.g.:
- Uninitialized Variables

Some analyses:

- Confluence:
  - $\bigcap = \cap$ (set intersection)

- Partial order:
  - $\supseteq = \supseteq$ (super-set-eq)

E.g.:
- Initialized Variables
- Available Expressions
- Very Busy Expressions

Analyze info that **may** possibly be true $\forall$ paths

Analyze info that **must** definitely be true $\forall$ paths
\[ \text{DUALITY: Lattice} \]

Lattice \( M \):

- Confluence: \( \cup_M \)
- Partial order: \( \subseteq_M \)

Lattice \( W \):

- Confluence: \( \cup_W = \cap_M \)
- Partial order: \( \subseteq_W = \supseteq_M \)
DUALITY: Framework

What we have done:

- Approximation:
  - "safe" to throw away info upwards

- With confluence:
  - $\cup$ (least upper bound)

- Least fixed point:
  - $\operatorname{lfp}(f) = \bigcup_{i \geq 0} f^i(\bot)$
  - Computed as:
    - $\bot \subseteq f(\bot) \subseteq f(f(\bot)) \subseteq \ldots$

- Greatest fixed point:
  - $\operatorname{gfp}(f) = \bigcap_{i \geq 0} f^i(\top)$
  - Computed as:
    - $\top \supseteq f(\top) \supseteq f(f(\top)) \supseteq \ldots$

"Stand on head":

- Approximation:
  - "safe" to throw away info downwards

- With confluence:
  - $\bigcap$ (greatest lower bound)
Agenda

- Quick recap (of everything so far):
  - "Putting it all together" → Data-Flow Analysis
- Fixed-Point Iteration Strategies (3x)
- "Sign Analysis"
- "Constant Propagation Analysis"
- "Initialized Variables Analysis"
- Set-Based Analysis Framework
- WORKSHOP
WORKSHOP

- **Reaching Definitions:**
  - '↓' (forward), '∪' (may / smallest set)

- **Live Variables:**
  - '↑' (backward), '∪' (may / smallest set)

- **Available Expressions:**
  - '↓' (forward), '∩' (must / largest set)

- **Very Busy Expressions:**
  - '↑' (backward), '∩' (must / largest set)
Reaching Definitions:

- The *reaching definitions* (for a given program point) are those assignments that *may have defined* the current *vals* of *vars*.

Example:

```c
int x = input;
int y;
while (x>1) {
    y = x / 2;
    if (y>2) x = x - y;
}
output y;
```
"Learning takes place through the active behavior of the student: it is what (s)he does that (s)he learns, not what the teacher does."

-- Ralph W. Tyler (1949)
1) Define the problem
2) Show that the problem is undecidable
3) Define a Lattice
   - Check that it is a lattice (and explain how)
4) Define monotone transfer functions
   - Check that they are monotone (and explain how)
5) Pick a program the analysis can analyze
   - Make a "The Entire Process" diagram (cf. slide #5)
6) Repeat 5) for program the analysis can’t…
7) Explain possible uses of the analysis
Now, please: 3’ recap

Please spend 3’ on thinking about and writing down the main ideas and points from the lecture – now!

**FIGURE 2.5. THE VALUE OF REHEARSAL FOLLOWING A LECTURE.**

<table>
<thead>
<tr>
<th>Percentage Recalled</th>
<th>Days from Lecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediately</td>
<td>After 3 weeks</td>
</tr>
<tr>
<td>After 2 weeks</td>
<td>After 1 week</td>
</tr>
<tr>
<td>After 1 day</td>
<td></td>
</tr>
</tbody>
</table>

*Source: Adapted from Bassey (1968).*