Program Analysis
Dr.-Ing. Florian Martin

Example
Input:
Pascal-like program with arrays
Output:
C-like program with address computation
Elimination of redundant computations
Simplifying address computations
Sequence of analyses and transformations

Input
a, b, c : array [0 : m, 0 : n] of integer

Output

C-Like Arrays

Analysis: Available Expressions

Recomputations

First computation

Line-wise array placement
Programanalyse

Common Subexpressions Eliminated

```plaintext
i := 0;
while i <= m do
  j := 0;
  while j <= n do
    t1 := i*(n+1)+j;
    ta := Base(a) + t1;
    C(ta):= C(Base(b) + t1) + C(Base(c) + t1);
    j := j + 1;
  i := i + 1;
```

Analysis: Loop Invariant Computation

```plaintext
i := 0;
while i <= m do
  j := 0;
  while j <= n do
    t1 := i*(n+1)+j;
    ta := Base(a) + t1;
    C(ta):= C(Base(b) + t1) + C(Base(c) + t1);
    j := j + 1;
  i := i + 1;
```

Loop Invariant Code Motion

```plaintext
i := 0;
while i <= m do
  j := 0;
  t2 := i*(n+1);
  while j <= n do
    t1 := t2 + j;
    ta := Base(a) + t1;
    C(ta):= C(Base(b) + t1) + C(Base(c) + t1);
    j := j + 1;
  i := i + 1;
```

Analysis: Induction Variable / Reaching Definitions

```plaintext
i := 0;
while i <= m do
  j := 0;
  t2 := i*(n+1);
  while j <= n do
    t1 := t2 + j;
    ta := Base(a) + t1;
    C(ta):= C(Base(b) + t1) + C(Base(c) + t1);
    j := j + 1;
  i := i + 1;
```

Strength Reduction

```plaintext
i := 0;
t3 := 0;
while i <= m do
  j := 0;
t2 := t3;
  while j <= n do
    t1 := t2 + j;
    ta := Base(a) + t1;
    C(ta):= C(Base(b) + t1) + C(Base(c) + t1);
    j := j + 1;
  t3 := t3 + (n+1);
  i := i + 1;
```

Copy Analysis

```plaintext
i := 0;
t3 := 0;
while i <= m do
  j := 0;
t2 := t3;
  while j <= n do
    t1 := t2 + j;
    ta := Base(a) + t1;
    C(ta):= C(Base(b) + t1) + C(Base(c) + t1);
    j := j + 1;
  t3 := t3 + (n+1);
  i := i + 1;
```
Copy Propagation

```c
i := 0;
t3 := 0;
while i <= m do
  ( j := 0; t2 := t3;
   while j <= n do
     t1 := t3 + j;
     ta := Base(a) + t1;
     C(ta):= C(Base(b) + t1) + C(Base(c) + t1);
     j := j + 1
   );
   t3 := t3 + (n+1);
   i := i + 1
);```

Analysis: Dead Variable

```c
i := 0;
t3 := 0;
while i <= m do
  ( j := 0; t2 := t3;
   while j <= n do
     t1 := t3 + j;
     ta := Base(a) + t1;
     C(ta):= C(Base(b) + t1) + C(Base(c) + t1);
     j := j + 1
   );
   t3 := t3 + (n+1);
   i := i + 1
);```

Elimination of Useless Assignment

```c
i := 0;
t3 := 0;
while i <= m do
  ( j := 0;
   while j <= n do
     t1 := t3 + j;
     ta := Base(a) + t1;
     C(ta):= C(Base(b) + t1) + C(Base(c) + t1);
     j := j + 1
   );
   t3 := t3 + (n+1);
   i := i + 1
);```

Overview

```
Motivation
Approximations
Data Flow Analysis
```

The Halting Problem

- Interesting problems are not computable
- Need of approximative solutions
- Only erring on the safe side allowed
- The safe side depends on interpretation of collected information
- Less erring means more precision

Exact Answers

```
Property holds
Property does not hold
```
Approximations

- Property *definitely holds*
- Property *might not hold*

Erring on the safe side

Overview

Data Flow Analysis

- Introduced ~1973
- Goal: Proving properties of a program for each program point
- Works on control flow graphs
- Correctness can be proved by abstract interpretation

Control Flow Graph

- Representation of program structure
- Nodes: Statements or statement parts
- Edges: Possible flow of control
- Edges can be labeled

Example: Control Flow Graph

\[
\begin{align*}
[y := x] ; \\
[z := 1] ; \\
\text{while}[y > 1] \forall \ (x := x \cdot y) ; \\
(y := y - 1) ; \\
\text{end} ; \\
(y := 0)
\end{align*}
\]
Overview

Control Flow Graph → Reaching Definitions Example → Equation Systems

Reaching Definitions

{ An assignment \([x := a]\) \(\rightarrow \) reaches \(k'\) if there is a path to \(k'\) on which the last assignment of \(x\) was in \(k\)

\[
\begin{align*}
\text{Example Equation Systems} & \\
\text{Systems} & \\
\end{align*}
\]

Reaching Definitions

\[
\begin{align*}
\text{System} 1: & \quad y := x \\
\text{System} 2: & \quad z := 1 \\
\text{System} 3: & \quad y := 0 \\
\end{align*}
\]

Reaching Definitions

\[
\begin{align*}
\text{System} 4: & \quad z := z \cdot y \\
\text{System} 5: & \quad y := y - 1 \\
\text{System} 6: & \quad y := 0 \\
\end{align*}
\]

Reaching Definitions

\[
\begin{align*}
\text{System} 7: & \quad y := x \\
\text{System} 8: & \quad z := 1 \\
\text{System} 9: & \quad y := 0 \\
\end{align*}
\]

Reaching Definitions

\[
\begin{align*}
\text{System} 10: & \quad z := z \cdot y \\
\text{System} 11: & \quad y := y - 1 \\
\text{System} 12: & \quad y := 0 \\
\end{align*}
\]

Reaching Definitions

\[
\begin{align*}
\text{System} 13: & \quad y := x \\
\text{System} 14: & \quad z := 1 \\
\text{System} 15: & \quad y := 0 \\
\end{align*}
\]

Reaching Definitions

\[
\begin{align*}
\text{System} 16: & \quad z := z \cdot y \\
\text{System} 17: & \quad y := y - 1 \\
\text{System} 18: & \quad y := 0 \\
\end{align*}
\]
Reaching Definitions

Another safe solution - but not the best:

\[ \begin{align*}
&[y := x] ; \\
&[z := 1] ; \\
&\text{while } [y>1] \\
&\text{do } ( [z := z*y] ; \\
&\quad [y := y-1] ) \\
&[y := 0] \\
\end{align*} \]

Overview

Computing Data Flow Information

- Extracting an equation system
- Computing the least fixpoint
Programanalyse

RD_{entry}(1) = (RD_{entry}(1) \setminus \{(y, l) | l \in Lab\}) \cup \{(y, 1)\}

RD_{exit}(1) = (RD_{entry}(1) \setminus \{(y, l) | l \in Lab\}) \cup \{(y, 1)\}

RD_{entry}(2) = (RD_{entry}(2) \setminus \{(z, l) | l \in Lab\}) \cup \{(z, 2)\}

RD_{exit}(2) = (RD_{entry}(2) \setminus \{(z, l) | l \in Lab\}) \cup \{(z, 2)\}

RD_{exit}(3) = RD_{entry}(3)

RD_{entry}(4) = (RD_{entry}(4) \setminus \{(z, l) | l \in Lab\}) \cup \{(z, 4)\}

RD_{exit}(4) = (RD_{entry}(4) \setminus \{(z, l) | l \in Lab\}) \cup \{(z, 4)\}

RD_{exit}(5) = (RD_{entry}(5) \setminus \{(y, l) | l \in Lab\}) \cup \{(y, 5)\}

RD_{exit}(5) = (RD_{entry}(5) \setminus \{(y, l) | l \in Lab\}) \cup \{(y, 5)\}

RD_{exit}(6) = (RD_{entry}(6) \setminus \{(y, l) | l \in Lab\}) \cup \{(y, 6)\}

RD_{exit}(6) = (RD_{entry}(6) \setminus \{(y, l) | l \in Lab\}) \cup \{(y, 6)\}
Programanalyse

\[
\begin{align*}
\text{RD}_\text{entry}(1) & \quad \text{RD}_\text{exit}(1) \\
[y := x] & ; \\
[z := 1] & ; \\
\text{while} \ [y > 1] & \text{do} \ ( [z := z \cdot y] ; \\
[y := y - 1] & ) \ ; \\
[y := 0] & \\
\text{RD}_\text{entry}(1) & \quad \text{RD}_\text{exit}(1)
\end{align*}
\]

\[
\begin{align*}
\text{RD}_\text{entry}(2) & \quad \text{RD}_\text{exit}(2) \\
\text{RD}_\text{entry}(3) & \quad \text{RD}_\text{exit}(3) \\
\text{RD}_\text{entry}(4) & \quad \text{RD}_\text{exit}(4) \\
\text{RD}_\text{entry}(5) & \quad \text{RD}_\text{exit}(5)
\end{align*}
\]

\[
\begin{align*}
\text{RD}_\text{entry}(1) & \quad \text{RD}_\text{exit}(1) \\
[y := x] & ; \\
[z := 1] & ; \\
\text{while} \ [y > 1] & \text{do} \ ( [z := z \cdot y] ; \\
[y := y - 1] & ) \ ; \\
[y := 0] & \\
\text{RD}_\text{entry}(6) & \quad \text{RD}_\text{exit}(6)
\end{align*}
\]

**Reaching Definitions: Summary**

- 12 sets: \( \text{RD}_{\text{entry}}(1), \ldots, \text{RD}_{\text{entry}}(6) \)
- 12 equations: 
  - \( \text{RD}_d = F (\text{RD}_{\text{entry}}(1), \ldots, \text{RD}_{\text{exit}}(6)) \)
- One function: 
  - \( F : (\wp(\text{Var} \times \text{Lab}))^2 \Rightarrow (\wp(\text{Var} \times \text{Lab}))^2 \)
- We want the least fixed point of \( F \): but does it exist?

**Overview**

- Equation Systems
- Lattice Theory
- Data Flow Analysis: Summary
Lattice Theory

A complete lattice \((L, \sqsubseteq)\) is a set \(L\) with a partial ordering \(\sqsubseteq\):
- Reflexive: \(l \sqsubseteq l\)
- Transitive: \(l_1 \sqsubseteq l_2 \land l_2 \sqsubseteq l_3 \Rightarrow l_1 \sqsubseteq l_3\)
- Anti-symmetric: \(l_1 \sqsubseteq l_2 \land l_2 \sqsubseteq l_1 \Rightarrow l_1 = l_2\)

and such that all subsets \(Y\) have least upper bounds \(\sqcup Y:\)
- Upper bound: \(\forall l \in Y: l \sqsubseteq \sqcup Y\)
- Least: \(\exists l \in Y: \forall l' \in Y: l' \sqsubseteq l\)

Lattice Theory: Lemma

- \(L\) is a complete lattice
- All subsets \(Y\) of \(L\) have least upper bounds \(\sqcup Y\)
- All subsets \(Y\) of \(L\) have greatest lower bounds \(\sqcap Y\)
- Lower bound: \(\forall l \in Y: \sqcap Y \sqsubseteq l\)
- Greatest: \(\exists l \in Y: \forall l' \in Y: l' \sqsubseteq l\)

Tarski's Theorem

\((L, \sqsubseteq), f: L \rightarrow L\)
- \(\text{Red}(f)\)
- \(\text{Fix}(f)\)
- \(\text{Ext}(f)\)
- \(\text{lfp}(f)\)
- \(\text{gfp}(f)\)
- \(\text{f}^0(\top)\)
- \(\text{f}^1(\top)\)
- \(\text{f}^2(\top)\)
- \(\text{...}\)
- \(\bot\)
### Chaotic Iteration

**Initialization:**
- RD₁ := Ø; … RD₁₂ := Ø;

**Iteration:**
- while RDₗ ≠ Fₗ(RD₁,...,RD₁₂) for some l
- do RDₗ := Fₗ(RD₁,...,RD₁₂)

### Chaotic Iteration: Correctness

**Initialization:**
- RD₁ := Ø; … RD₁₂ := Ø;

**Iteration:**
- while RDₗ ≠ Fₗ(RD₁,...,RD₁₂) for some l
- do RDₗ := Fₗ(RD₁,...,RD₁₂)
- If the algorithm terminates
  - then it computes the least fixed point of F:
    - Fⁿ⁺¹(Ø) = Fⁿ(Ø) = lfp(F)

### Chaotic Iteration: Termination

**Initialization:**
- RD₁ := Ø; … RD₁₂ := Ø;

**Iteration:**
- while RDₗ ≠ Fₗ(RD₁,...,RD₁₂) for some l
- do RDₗ := Fₗ(RD₁,...,RD₁₂)
- The algorithm terminates because
  - RDₗ ∈ F(RD₁,...,RD₁₂)
  - is only possible finitely many times
  - since (℘((Var* × Lab*))¹² is finite

### Overview

**Data Flow Analysis**

- Complete lattice with ascending chain property
- Equation system
- Transfer functions
  - combine function
- Control flow graph
  - Least fixed point exists and is computable

**Add-ons:**
- Direction (forward, backward)
- Least/greatest fixed point
- Initialization
- Value of the start node
Overview

Data Flow Analysis Summary

Some Examples

Exercises

Classical Analyses

<table>
<thead>
<tr>
<th></th>
<th>Union</th>
<th>Intersection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward</td>
<td>RD</td>
<td>AE</td>
</tr>
<tr>
<td>Backward</td>
<td>LV</td>
<td>VB</td>
</tr>
</tbody>
</table>

Reaching Definitions

Problem: for each program point, which assignments may have been made and not overwritten, when program execution reaches this point along some path.

Example: 
\[ x := 5; y := 1; \text{ while } x > 1 \text{ do } ( y := x; x := x - 1) \]

Idea:

\[ N = N_1 \cup N_2 \]

\[ X = N \setminus \text{kill}(B) \cup \text{gen}(B) \]

Available Expressions

Problem: for each program point, which expressions must have already been computed and not later modified, on all paths reaching the program point.

Example: 
\[ x := a + b; y := a \ast b; \text{ while } y > a \ast b \text{ do } (a := a + 1; x := a + b) \]

Idea:

\[ N = N_1 \cap N_2 \]

\[ X = N \setminus \text{kill}(B) \cup \text{gen}(B) \]

Reaching Definitions

\[ \text{gen}(z := a) = \{ (x, a) \} \]

\[ \text{kill}(x := a) = \{ (x, x') \mid x \text{ is in the program } \} \]

\[ \text{RD}_{\text{com}}(B) = \begin{cases} (x, y) \mid x \text{ is in the program } & \text{if } B \text{ is "enter"} \\ \bigcup (\text{RD}_{\text{com}}(B) \uparrow B \rightarrow B \text{ in the program}) & \text{otherwise} \end{cases} \]

\[ \text{RD}_{\text{com}}(B) = \text{RD}_{\text{com}}(B) \setminus \text{kill}(B) \cup \text{gen}(B) \]

Available Expressions

\[ \text{gen}(z := a) = \{ (x, a) \mid a \text{ subexpression of } a \mid a \text{ in } a' \} \]

\[ \text{kill}(x := a) = \{ (x, x) \mid x \text{ occurs in } a' \} \]

\[ \text{AE}_{\text{com}}(B) = \begin{cases} (\text{AE}_{\text{com}}(B) \uparrow B \rightarrow B \text{ in the program}) & \text{if } B \text{ is "enter"} \\ \emptyset & \text{otherwise} \end{cases} \]

\[ \text{AE}_{\text{com}}(B) = \text{AE}_{\text{com}}(B) \setminus \text{kill}(B) \cup \text{gen}(B) \]
Live Variables

Problem: for each program point, which variables may be live at the exit from that point — a variable is live at the exit of a program point if there is some path from the point that uses the variable before it becomes not live.

Example:

\[ x := 2; y := 4; z := y \Rightarrow x \text{ if } y > x \text{ then } z := y \text{ else } z := y; y := x; x := z \]

Idea:

\[ B \]

\[ N_1 \quad N_2 \quad \]

\[ N_3 \]

Live Variables

\[ \text{kill}(\exists x := a) = \{ x \} \]

\[ \text{gen}(\exists x := a) = \{ x \mid x \text{ occurs in } a \} \]

\[ \text{LV}_{\text{expr}}(B) = \begin{cases} \emptyset & \text{if } B \text{ is "exit"} \\ \bigcup \{ \text{LV}_{\text{expr}}(B') \mid B' \text{ is in the program} \} & \text{otherwise} \end{cases} \]

\[ \text{LV}_{\text{expr}}(B) = \text{LV}_{\text{expr}}(B) \cup \text{kill}(B) \cup \text{gen}(B) \]

Very Busy Expressions

Problem: for each program point, which expressions must be very busy at the exit from that point — an expression is very busy at the exit of a program point if it is used on all paths from that point before any of its variables are redefined.

Example:

\[ \text{if } a > b \text{ then } (x := a + b; y := a - b) \text{ else } (x := a + b; x := a - b) \]

Idea:

\[ N_1 \quad N_2 \quad N_3 \]

Very Busy Expressions

\[ \text{gen}(\exists x := a) = \{ a \mid a \text{ is subexpression of } a \} \]

\[ \text{kill}(\exists x := a) = \{ a \mid x \text{ occurs in } a \} \]

\[ \text{VB}_{\text{expr}}(B) = \begin{cases} \emptyset & \text{if } B \text{ is "exit"} \\ \bigcup \{ \text{VB}_{\text{expr}}(B') \mid B' \text{ is in the program} \} & \text{otherwise} \end{cases} \]

\[ \text{VB}_{\text{expr}}(B) = \text{LV}_{\text{expr}}(B) \cup \text{kill}(B) \cup \text{gen}(B) \]

Restrictions

\{ CFC needed \Rightarrow \text{imperative languages only} \}

Additional Reading

\{ Nielson, Nielson, Hankin: \textit{Principles of Program Analysis} \\
Springer \}

\{ PAG/WWW \}

http://www.program-analysis.com