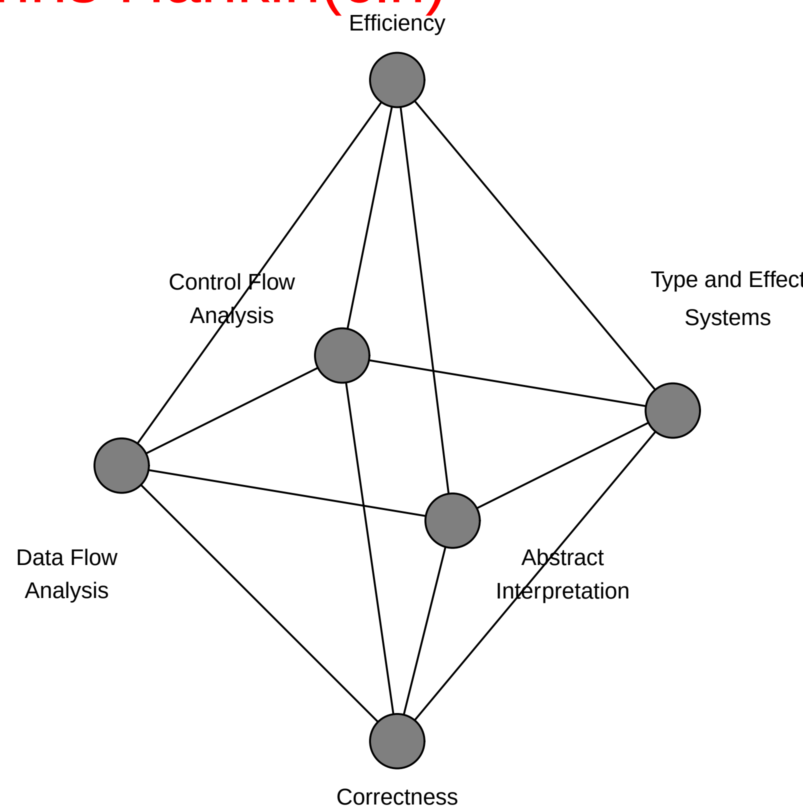
Program Analysis

Chris Hankin(clh)



Overview

- Introduction
- Data Flow Analysis
- Control Flow Analysis
- Algorithms

Introduction

Program analysis is an automatic technique for finding out properties of programs without having to run them.

- Optimising compilers
- Automated program verification
- Security

Some techniques:

- Data Flow Analysis
- Control Flow Analysis
- Types and Effects Systems
- Abstract Interpretation

Book: *Principles of Program Analysis* by F. Nielson, H.R. Nielson and C. Hankin, Springer Verlag, 1999.

A first example:

$$[\textbf{input } n]^1;$$

$$[m := 2]^2;$$

$$\textbf{while } [n > 1]^3 \textbf{ do}$$

$$[m := m \times n]^4;$$

$$[n := n - 1]^5;$$

$$[\textbf{output } m]^6;$$

We can statically determine that the value of m at statement 6 will be even for any input n. A program analysis can determine this by propagating parity information *forwards* from the start of the program.

We can assign one of three properties to each variable:

- even – the value is known to be even
- odd – the value is known to be odd
- unknown – the parity of the value is unknown

(Take care of loop)

1:	m: unknown	n: unknown
2:	m: unknown	n: unknown
3:	m: even	n: unknown
4:	m: even	n: unknown
5:	m: even	n: unknown
6:	m: even	n: unknown

The program computes 2 times the factorial of n for any positive value of n. Replacing statement 2 by:

$$[m := 1]^2;$$

gives a program that computes factorials but then the program analysis is unable to tell us anything about the parity of m at statement 6.

This is correct because m could be even or odd. However, even if we fix the input to be positive and even, by some suitable conditional assignment, the program analysis will still not accurately predict the evenness of m at statement 6.

This loss of accuracy is a common feature of program analyses: many properties that we are interested in are essentially *undecidable* and therefore we cannot hope to detect them accurately. We have to ensure that the answers from program analysis are at least *safe*.

- **yes** means definitely yes, and
- **no** means <u>possibly</u> no.

In the modified factorial program, it is safe to say that the parity of m is unknown at 6 – it would not be safe to say that m is even.

We identify three facets of program analysis:

- specification,
- efficient implementations, and
- correctness

- The starting point for data flow analysis is some representation of the control flow graph of the programs.
- The Data Flow Analysis is usually specified as a set of equations which associate analysis information with program points. Program points correspond to nodes in the graph.
- Analysis information may be propagated forwards through the program, as in the parity analysis, or backwards.
- When the control flow graph is not explicitly given, we need a preliminary Control Flow Analysis.

Reaching Definitions determines which set of definitions (assignments) are current when control reaches a certain program point. The analysis can be specified by equations of the following form:

$$
\begin{array}{lcll}
RD_{entry}(p) & = & \begin{cases} \iota & \text{if } p \text{ is initial} \\ \bigcup_{p' \in pred(p)} RD_{exit}(p') & \text{otherwise} \end{cases} \\
RD_{exit}(p) & = & (RD_{entry}(p) \backslash kill(p)) \ \cup \ gen(p)
\end{array}
$$

- Each program point kills some definitions (those which define the same variable as the program point) and generates new definitions.
- A suitable representation for properties is sets of pairs where each pair is a variable and a program point – (x, p). The initial value in this case is:

$$\iota = \{(x, ?) \mid x \text{ is a variable in the program}\}$$

- Reaching Definitions is a forwards analysis.

$$
\begin{aligned}
RD_{entry}(1) &= \{(m, ?), (n, ?)\} \\
RD_{entry}(3) &= RD_{exit}(2) \cup RD_{exit}(5)
\end{aligned}
$$

	RD_{entry}	RD_{exit}
1	$\{(m,?),(n,?)\}$	$\{(m,?),(n,1)\}$
2	$\{(m,?),(n,1)\}$	$\{(m,2),(n,1)\}$
3	$\{(m,2),(m,4),(n,1),(n,5)\}$	$\{(m,2),(m,4),(n,1),(n,5)\}$
4	$\{(m,2),(m,4),(n,1),(n,5)\}$	$\{(m,4),(n,1),(n,5)\}$
5	$\{(m,4),(n,1),(n,5)\}$	$\{(m,4),(n,5)\}$
6	$\{(m,2),(m,4),(n,1),(n,5)\}$	$\{(m,2),(m,4),(n,1),(n,5)\}$

INPUT: A control flow graph

OUTPUT: RD

METHOD: Step 1: Initialisation

for all program points, p do

RD(p) := $\emptyset$;

RD(1) := ι;

Step 2: Iteration

```
change := true;
while change do
    change := false;
    for all program points, p do
        new := ⋃_{p' ∈ pred(p)} f(RD,p')
        if RD(p) ≠ new then
            change := true;
            RD(p) := new;
```

Where: $\text{new} := \bigcup_{p' \in pred(p)} f(RD,p')$

USING: $f(RD,p) = (RD(p) \setminus kill(p)) \cup gen(p);$

Some example data flow analyses:

1. Reaching Definitions – Constant Folding
2. Available Expressions – Avoiding recomputation
3. Very Busy Expressions – Hoisting
4. Live Variables – Dead Code Elimination
5. Information Flow – No Read-up, No Write-down

To illustrate the ideas we shall show how Reaching Definitions can be used to perform Constant Folding. There are two ingredients in this:

- One is to replace the use of a variable in some expression by a constant if it is known that the value of the variable will always be that constant.
- The other is to simplify an expression by partially evaluating it: subexpressions that contain no variables can be evaluated.

$$\mathsf{RD} \vdash [x := a]^\ell \ \triangleright\ [x := a[y \mapsto n]]^\ell$$

$$\text{if} \begin{cases} y \in \mathit{FV}(a) \ \wedge\ (y, ?) \notin \mathsf{RD}_{entry}(\ell) \ \wedge \\ \forall (z, \ell') \in \mathsf{RD}_{entry}(\ell) : (z = y \Rightarrow [\cdots]^{\ell'} \text{ is } [y := n]^{\ell'}) \end{cases}$$

$$\mathsf{RD} \vdash [x := a]^\ell \ \triangleright\ [x := n]^\ell$$

$$\text{if } \mathit{FV}(a) = \emptyset \ \wedge\ a \notin \mathbf{Num} \ \wedge\ a \text{ evaluates to } n$$

$$\frac{\mathsf{RD} \vdash S_1 \ \triangleright\ S_1'}{\mathsf{RD} \vdash S_1; S_2 \ \triangleright\ S_1'; S_2}$$

$$\frac{\mathsf{RD} \vdash S_2 \ \triangleright\ S_2'}{\mathsf{RD} \vdash S_1;S_2 \ \triangleright\ S_1;S_2'}$$

$$\frac{\mathsf{RD} \vdash S_1 \ \triangleright\ S_1'}{\mathsf{RD} \vdash \texttt{if } [b]^\ell \texttt{ then } S_1 \texttt{ else } S_2 \ \triangleright\ \texttt{if } [b]^\ell \texttt{ then } S_1' \texttt{ else } S_2}$$

$$\frac{\mathsf{RD} \vdash S_2 \ \triangleright\ S_2'}{\mathsf{RD} \vdash \texttt{if } [b]^\ell \texttt{ then } S_1 \texttt{ else } S_2 \ \triangleright\ \texttt{if } [b]^\ell \texttt{ then } S_1 \texttt{ else } S_2'}$$

$$\frac{\mathsf{RD} \vdash S \ \triangleright\ S'}{\mathsf{RD} \vdash \texttt{while } [b]^\ell \texttt{ do } S \ \triangleright\ \texttt{while } [b]^\ell \texttt{ do } S'}$$

To illustrate the use of the transformation consider the program:

$$[x := 10]^1;[y := x + 10]^2;[z := y + 10]^3$$

A solution to the Reaching Definitions Analysis for this program is:

$$\begin{array}{rcl}
\mathsf{RD}_{entry}(1) & = & \{(x,?),(y,?),(z,?)\} \\
\mathsf{RD}_{exit}(1) & = & \{(x,1),(y,?),(z,?)\} \\
\mathsf{RD}_{entry}(2) & = & \{(x,1),(y,?),(z,?)\} \\
\mathsf{RD}_{exit}(2) & = & \{(x,1),(y,2),(z,?)\} \\
\mathsf{RD}_{entry}(3) & = & \{(x,1),(y,2),(z,?)\} \\
\mathsf{RD}_{exit}(3) & = & \{(x,1),(y,2),(z,3)\}
\end{array}$$

We can obtain the following transformation sequence:

$$
\begin{array}{lll}
\text{RD} & \vdash & [x := 10]^1;[y := x + 10]^2;[z := y + 10]^3 \\
& \rhd & [x := 10]^1;[y := 10 + 10]^2;[z := y + 10]^3 \\
& \rhd & [x := 10]^1;[y := 20]^2;[z := y + 10]^3 \\
& \rhd & [x := 10]^1;[y := 20]^2;[z := 20 + 10]^3 \\
& \rhd & [x := 10]^1;[y := 20]^2;[z := 30]^3
\end{array}
$$

after which no more steps are possible.

The above example shows that we shall want to perform many successive transformations:

$$\mathsf{RD} \vdash S_1 \rhd S_2 \rhd \cdots \rhd S_{n+1}$$

This could be costly because once S_1 has been transformed into S_2 we might have to *recompute* Reaching Definitions Analysis for S_2 before the transformation can be used to transform it into S_3 etc. It turns out that it is sometimes possible to use the analysis for S_1 to obtain a reasonable analysis for S_2 without performing the analysis from scratch.