Introduction to Program Analysis

Herbert Wiklicky
herbert@doc.ic.ac.uk
www.doc.ic.ac.uk/∼herbert

Department of Computing
Imperial College London

Verona, April 2012

Static Program Analysis

Program Analysis is an automated technique for determining properties of programs without having to execute them.
We can distinguish in particular between:

Static Analysis vs **Dynamic Testing**

The results obtained by static program analysis can be used in:

- Compiler Optimisation
- Program Verification
- Security Analysis

Techniques

The techniques used in program analysis include e.g.:

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

A comprehensive introduction and details can be found in:

Flemming Nielson, Hanne Riis Nielson and Chris Hankin: *Principles of Program Analysis*. Springer Verlag, 1999/2005.

A First Example

Consider the following fragment in *some* procedural language.

$m \leftarrow 2;$
while $n > 1$ **do**
    $m \leftarrow m \times n;$
    $n \leftarrow n - 1$
end while
stop

$[m \leftarrow 2]^1;$
while $[n > 1]^2$ **do**
    $[m \leftarrow m \times n]^3;$
    $[n \leftarrow n - 1]^4$
end while
$[\textbf{stop}]^5$

We annotate a program such that it becomes clear about what program point p or label ℓ we are talking about. This annotation can easily be defined formally.

A Parity Analysis

Claim: This program fragment always returns an **even** m, independently of the initial values of m and n.

We can statically determine that in any circumstances the value of m at the last statement will be **even** for any input n.

A program analysis, so-called parity analysis, can determine this property of the program by propagating the even/odd or *parity* information *forwards* form the start of the program.

Properties

We will assign to each variable one of three properties:

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

For both variables m and n we record their parity at each stage of the computation (i.e. we investigate at the computational situation at the beginning of each statement).

A First Example

Executing the program with *abstract* values – parity – for m and n results in the following:

$m \leftarrow 2$; ▷ unknown(m) – unknown(n)
while $n > 1$ **do** ▷ even(m) – unknown(n)
  $m \leftarrow m \times n$; ▷ even(m) – unknown(n)
  $n \leftarrow n - 1$ ▷ even(m) – unknown(n)
end while ▷ even(m) – unknown(n)
stop ▷ even(m) – unknown(n)

Important: We can restart the loop with the same information about the parity of m and n over and over again!

A First Example

The first program computes 2 times the factorial for any positive value of n. Replacing '2' by '1' in the first statement gives:

$m \leftarrow 1$; ▷ unknown(m) – unknown(n)
while $n > 1$ **do** ▷ unknown(m) – unknown(n)
  $m \leftarrow m \times n$; ▷ unknown(m) – unknown(n)
  $n \leftarrow n - 1$ ▷ unknown(m) – unknown(n)
end while ▷ unknown(m) – unknown(n)
stop ▷ unknown(m) – unknown(n)

i.e. the plain factorial – but in this case the program analysis is unable to tell us anything about the parity of m at the end of the execution.

Loss of Precision

The analysis of the new program, i.e. the plain factorial, does not give any satisfying result because:

- m could be **even** — if the input $n > 1$, or
- m could be **odd** — if the input $n \leq 1$.

However, even if we fix/require the input to be positive and **even** — e.g. by some suitable conditional assignment — the program analysis still might not be able to accurately predict that m will be **even** at statement **6**.

Alternative: Perform a probabilistic program analysis.

Safe Approximations

Such a loss of precession is a common feature of program analysis: Many properties that we are interested in are essentially undecidable and therefore we cannot hope to detect (all of) them accurately.

We only aim to ensure that the answers/results we obtain by program analysis are at least safe, i.e.

- yes means *definitely* yes,
- no means *maybe* no.

It is necessary to always have a result (i.e. the analysis terminates) but we have to accept that this result is **unknown**.

Facets of Program Analysis

We can identify the following facets of program analysis which play a role when considering a particular program property:

- Specification
- Implementation
- Correctness
- Applications

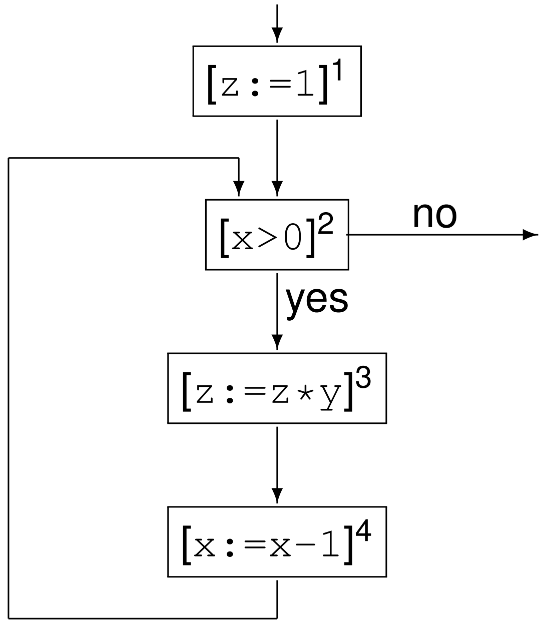
Data Flow Analysis

The starting point for data flow analysis is a representation of the control flow graph of the program: the nodes of such a graph may represent individual statements – as in a flowchart – or sequences of statements; arcs specify how control may be passed during program execution.

The data flow analysis is usually specified as a set of equations which associate analysis information with program points which correspond to the nodes in the control flow graph. This information may be propagated *forwards* through the program (e.g. parity analysis) or *backwards*.

When the control flow graph is not explicitly given, we need a preliminary control flow analysis

Control Flow Information



This allows us to determine the predecessors *pred* and successors *succ* of each statement, e.g. $pred(2) = \{1, 4\}$.

Flow for WHILE

Statements $S \in$ **Stmt** have the abstract (labelled) syntax:

$$S ::= [x{:=}a]^\ell \mid [\textbf{skip}]^\ell \mid S_1;S_2 \mid \textbf{if } [b]^\ell \textbf{ then } S_1 \textbf{ else } S_2 \mid \textbf{while } [b]^\ell \textbf{ do } S$$

with a arithmetic, b boolean expressions and labels $\ell \in$ **Lab**. Blocks $B \in$ **Block** are of the form: $[\ x := a\ ]^\ell$, $[\ \textbf{skip}\ ]^\ell$, or $[b]^\ell$.

Formally define for all (composite) statements how to extract the initial and final labels:

$$init : \textbf{Stmt} \to \textbf{Lab} \qquad final : \textbf{Stmt} \to \mathcal{P}(\textbf{Lab})$$

as well as the possible control steps between labels:

$$flow : \textbf{Stmt} \to \mathcal{P}(\textbf{Lab} \times \textbf{Lab})$$

An Example Flow

Consider the following program, power, computing the x-th power of the number stored in y:

$$
\begin{array}{l}
[\ z := 1\ ]^1; \\
\textbf{while}\ [x > 1]^2\ \textbf{do}\ (\\
\qquad [\ z := z * y\ ]^3; \\
\qquad [\ x := x - 1\ ]^4);
\end{array}
$$

We have *init*(power) $= 1$, and *final*(power) $= \{2\}$. The function *flow* produces the set:

$$\mathit{flow}(\text{power}) = \{(1,2),(2,3),(3,4),(4,2)\}$$

Reaching Definition

Reaching Definition (*RD*) analysis determines which set of definitions (i.e. assignments) are current when control reaches a certain program point ℓ.

The analysis can be specified by equations of the form:

$$\mathrm{RD}_{entry}(\ell) = \begin{cases} \{(x,?) \mid x \in FV(S_\star)\}, & \text{if } \ell = init(S_\star) \\ \bigcup\{\mathrm{RD}_{exit}(\ell') \mid (\ell',\ell) \in flow(S_\star)\}, & \text{otherwise} \end{cases}$$

$$\begin{array}{rcl} \mathrm{RD}_{exit}(\ell) & = & (\mathrm{RD}_{entry}(\ell) \backslash kill_{\mathrm{RD}}([B]^\ell)) \cup gen_{\mathrm{RD}}([B]^\ell) \\ & & \text{where } [B]^\ell \in blocks(S_\star) \end{array}$$

Analysis Information

At each program point some definitions get "killed" (those which define the same variable as at the program point) while others are "generated".

A suitable representation for properties are sets of pairs, where each pair contains a variable x and a program point ℓ: the meaning of the pair (x, ℓ) is that the assignment to x at point ℓ is the current one. The initial value in this case is:

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

Reaching Definitions is a forward analysis and we require the least (most precise) solutions to the set of equations.

Local Analysis

New analysis information is created depending on the kind of block are considering:

$$\begin{aligned} gen_{\text{RD}}([\, x := a \,]^\ell) &= \{(x, \ell)\} \\ gen_{\text{RD}}([\, \textbf{skip} \,]^\ell) &= \emptyset \\ gen_{\text{RD}}([b]^\ell) &= \emptyset \end{aligned}$$

Information about which (previous) "definitions" $[\, x := \ldots \,]^\ell$ are no longer current is constructed using:

$$\begin{aligned} kill_{\text{RD}}([\, x := a \,]^\ell) &= \{(x, ?)\} \cup \{(x, \ell') \mid \\ &\qquad [B]^{\ell'} \text{ a "definition" of } x \text{ in } S_\star\} \\ kill_{\text{RD}}([\, \textbf{skip} \,]^\ell) &= \emptyset \\ kill_{\text{RD}}([b]^\ell) &= \emptyset \end{aligned}$$

Equations & Solutions

For our initial program fragment

$[m \leftarrow 2]^1;$
while $[n > 1]^2$ **do**
  $[m \leftarrow m \times n]^3;$
  $[n \leftarrow n - 1]^4$
end while
[**stop**]^5

some of the *RD* equations we get are:

$$\begin{aligned} \mathrm{RD}_{entry}(1) &= \{(m, ?), (n, ?)\} \\ \mathrm{RD}_{entry}(2) &= \mathrm{RD}_{exit}(1) \cup \mathrm{RD}_{exit}(4) \end{aligned}$$

Equations & Solutions

$$\begin{aligned} \mathrm{RD}_{entry}(1) &= \{(m, ?), (n, ?)\} \\ \mathrm{RD}_{entry}(2) &= \mathrm{RD}_{exit}(1) \cup \mathrm{RD}_{exit}(4) \end{aligned}$$

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

Solving Equations

How can we construct solution to the data flow equations?
Answer: Iteratively, by improving approximations/guesses.

INPUT: Control Flow Graph
i.e. initial(p), pred(p).

OUTPUT: Reaching Definitions *RD*.

METHOD: Step 1: Initialisation
Step 2: Iteration

Data Flow Analysis

The general approach for determining program properties for procedural languages via a dataflow analysis:

- Extract Data Flow Information
- Formulate Data Flow Equations
  - Update Local Information
  - Collect Global Information
- Construct Solution(s) of Equations

Some other Analyses

Examples of data flow analyses — and the possible applications and optimisations they allow for — are:

- Reaching Definitions — Constant Folding
- Available Expressions — Avoid Re-computations
- Very Busy Expressions — Hoisting
- Live Variables — Dead Code Elimination
- Shape Analysis — Pointer Analysis
- Information Flow — Computer Security
- etc. etc.

Code Optimisation

To illustrate the ideas we shall show how Reaching Definitions can be used to perform Constant Folding.

There are two ingredients to this:

- Replace the use of a variable in some expression by a constant if it is known that the value of that variable will always be a constant.
- Simplify an expression by partially evaluating it: subexpressions that contain no variables can be evaluated.

Constant Folding I

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

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

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

$$\text{if} \begin{cases} FV(a) = \emptyset \;\wedge\; a \text{ is not constant} \;\wedge \\ a \text{ evaluates to } n \end{cases}$$

Constant Folding II

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

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

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

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

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

An Example

To illustrate the use of the transformation consider:

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

The (least) solution to the Reaching Definition analysis is:

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

Constant Folding

We have for example the following:

$$RD \vdash [\, y := x + 10\,]^2 \;\triangleright\; [\, y := 10 + 10\,]^2$$

and therfore the rules for sequential composition allow us to do the following transformation:

$$\begin{array}{ll}
RD \vdash & [\, x := 10\,]^1; [\, y := x + 10\,]^2; [\, z := y + 10\,]^3 \;\triangleright \\
& [\, x := 10\,]^1; [\, y := 10 + 10\,]^2; [\, z := y + 10\,]^3
\end{array}$$

Transformation

We can continue this kind of transformation and obtain:

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

after which no more steps are possible.

Additional Issues

The above example shows that optimisation is in general the result of a number of successive transformations.

$$RD \vdash S_1 \;\rhd\; S_2 \;\rhd\; \ldots \;\rhd\; S_n.$$

This could be costly because one S_1 has been transformed into S_2 we might have to *re-compute* the Reaching Definition analysis before the next transformation step can be done.

It could also be the case that different sequences of transformations either lead to different end results or are of very different length.

Designing an Analysis

- Identify property space:
  Very often $\mathcal{P}(X)$ or more general a lattice which specifies which information is more precise and which is less; as well as how to combine information. Eg *Parity* $= \mathcal{P}(\textit{even}, \textit{odd})$.
- Specify analysis (transformations/equations/constraints):
  State rules, e.g. using so-called monotone framework, on how properties change when a certain computational step happens, eg. *even* $\times$ *unknown* $=$ *even*.
- Address correctness and efficency/termination:
  Proof, using a formal semantics, that the rules are correct or constructed in a guaranteed safe way, e.g. using Abstract Interpretation. Improve and accelerate approximation process for finding solutions, e.g. using widening, etc.

Extra Slides

Initial Label

When presenting examples of Data Flow Analyses we will use a number of operations on programs and labels. The first of these is

$$\mathit{init} : \mathbf{Stmt} \to \mathbf{Lab}$$

which returns the initial label of a statement:

$$\begin{aligned}
\mathit{init}([\, x := a \,]^{\ell}) &= \ell \\
\mathit{init}([\, \mathbf{skip} \,]^{\ell}) &= \ell \\
\mathit{init}(S_1; S_2) &= \mathit{init}(S_1) \\
\mathit{init}(\mathbf{if}\ [b]^{\ell}\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2) &= \ell \\
\mathit{init}(\mathbf{while}\ [b]^{\ell}\ \mathbf{do}\ S) &= \ell
\end{aligned}$$

Final Labels

We will also need a function which returns the set of final labels in a statement; whereas a sequence of statements has a single entry, it may have multiple exits (e.g. in the conditional):

$$\mathit{final} : \mathbf{Stmt} \to \mathcal{P}(\mathbf{Lab})$$

$$\begin{aligned}
\mathit{final}([\, x := a \,]^{\ell}) &= \{\ell\} \\
\mathit{final}([\, \mathbf{skip} \,]^{\ell}) &= \{\ell\} \\
\mathit{final}(S_1; S_2) &= \mathit{final}(S_2) \\
\mathit{final}(\mathbf{if}\ [b]^{\ell}\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2) &= \mathit{final}(S_1) \cup \mathit{final}(S_2) \\
\mathit{final}(\mathbf{while}\ [b]^{\ell}\ \mathbf{do}\ S) &= \{\ell\}
\end{aligned}$$

Note that the **while**-loop terminates just after the test fails.

Flow

$$flow : \textbf{Stmt} \to \mathcal{P}(\textbf{Lab} \times \textbf{Lab})$$

maps statements to sets of flows:

$$\begin{array}{rcl}
flow([\, x := a \,]^\ell) & = & \emptyset \\
flow([\, \textbf{skip} \,]^\ell) & = & \emptyset \\
flow(S_1 ; S_2) & = & flow(S_1) \cup flow(S_2) \cup \\
 & & \{(\ell, init(S_2)) \mid \ell \in final(S_1)\} \\
flow(\textbf{if } [b]^\ell \textbf{ then } S_1 \textbf{ else } S_2) & = & flow(S_1) \cup flow(S_2) \cup \\
 & & \{(\ell, init(S_1)), (\ell, init(S_2))\} \\
flow(\textbf{while } [b]^\ell \textbf{ do } S) & = & flow(S) \cup \{(\ell, init(S))\} \cup \\
 & & \{(\ell', \ell) \mid \ell' \in final(S)\}
\end{array}$$

RD Example

A simple example to illustrate the *RD* analysis:

$$\begin{array}{l}
[\, x := 5 \,]^1; \\
[\, y := 1 \,]^2; \\
\textbf{while } [x > 1]^3 \textbf{ do } (\\
\qquad [\, y := x * y \,]^4; \\
\qquad [\, x := x - 1 \,]^5 \;)
\end{array}$$

All of the assignments reach the entry of 4 (the assignments labelled 1 and 2 reach there on the first iteration); only the assignments labelled 1, 4 and 5 reach the entry of 5.

RD Example: Local Information

$$
\begin{array}{l}
[\ x := 5\ ]^1; \\
[\ y := 1\ ]^2; \\
\textbf{while } [x > 1]^3 \textbf{ do } (\\
\qquad [\ y := x * y\ ]^4; \\
\qquad [\ x := x - 1\ ]^5\ )
\end{array}
$$

ℓ	$kill_{\mathrm{RD}}(\ell)$	$gen_{\mathrm{RD}}(\ell)$
1	$\{(x,?),(x,1),(x,5)\}$	$\{(x,1)\}$
2	$\{(y,?),(y,2),(y,4)\}$	$\{(y,2)\}$
3	$\emptyset$	$\emptyset$
4	$\{(y,?),(y,2),(y,4)\}$	$\{(y,4)\}$
5	$\{(x,?),(x,1),(x,5)\}$	$\{(x,5)\}$

RD Example: Equations (Entry)

$$
\begin{array}{l}
[\ x := 5\ ]^1; \\
[\ y := 1\ ]^2; \\
\textbf{while } [x > 1]^3 \textbf{ do } (\\
\qquad [\ y := x * y\ ]^4; \\
\qquad [\ x := x - 1\ ]^5\ )
\end{array}
$$

$$
\begin{array}{lcl}
\mathrm{RD}_{entry}(1) & = & \{(x,?),(y,?)\} \\
\mathrm{RD}_{entry}(2) & = & \mathrm{RD}_{exit}(1) \\
\mathrm{RD}_{entry}(3) & = & \mathrm{RD}_{exit}(2) \cup \mathrm{RD}_{exit}(5) \\
\mathrm{RD}_{entry}(4) & = & \mathrm{RD}_{exit}(3) \\
\mathrm{RD}_{entry}(5) & = & \mathrm{RD}_{exit}(4)
\end{array}
$$

RD Example: Equations (Exit)

$$
\begin{array}{l}
[\; x := 5 \;]^1; \\
[\; y := 1 \;]^2; \\
\textbf{while } [x > 1]^3 \textbf{ do } (\\
\qquad [\; y := x * y \;]^4; \\
\qquad [\; x := x - 1 \;]^5 \;)
\end{array}
$$

$$
\begin{array}{rcl}
\mathrm{RD}_{exit}(1) & = & (\mathrm{RD}_{entry}(1) \backslash \{(x,?),(x,1),(x,5)\}) \cup \{(x,1)\} \\
\mathrm{RD}_{exit}(2) & = & (\mathrm{RD}_{entry}(2) \backslash \{(y,?),(y,2),(y,4)\}) \cup \{(y,2)\} \\
\mathrm{RD}_{exit}(3) & = & \mathrm{RD}_{entry}(3) \\
\mathrm{RD}_{exit}(4) & = & (\mathrm{RD}_{entry}(4) \backslash \{(y,?),(y,2),(y,4)\}) \cup \{(y,4)\} \\
\mathrm{RD}_{exit}(5) & = & (\mathrm{RD}_{entry}(5) \backslash \{(x,?),(x,1),(x,5)\}) \cup \{(x,5)\}
\end{array}
$$

RD Example: Solutions

$$
\begin{array}{l}
[\; x := 5 \;]^1; \\
[\; y := 1 \;]^2; \\
\textbf{while } [x > 1]^3 \textbf{ do } (\\
\qquad [\; y := x * y \;]^4; \\
\qquad [\; x := x - 1 \;]^5 \;)
\end{array}
$$

ℓ	$\mathrm{RD}_{entry}(\ell)$	$\mathrm{RD}_{exit}(\ell)$
1	$\{(x,?),(y,?)\}$	$\{(y,?),(x,1)\}$
2	$\{(y,?),(x,1)\}$	$\{(x,1),(y,2)\}$
3	$\{(x,1),(y,2),(y,4),(x,5)\}$	$\{(x,1),(y,2),(y,4),(x,5)\}$
4	$\{(x,1),(y,2),(y,4),(x,5)\}$	$\{(x,1),(y,4),(x,5)\}$
5	$\{(x,1),(y,4),(x,5)\}$	$\{(y,4),(x,5)\}$