Dataflow analysis

Michel Schinz
Advanced Compiler Construction – 2009-05-01

A first example: Available expressions

Common subexp. elimination

The following C program fragment sets r to x^y for $y > 0$. How can it be (slightly) optimised?

```
 1 int y1 = 1;
 2 int r = x;
 3 while (y1 != y) {
 4   int t = y1*2;
 5   if (t <= y) {
 6     r = r*r;
 7     y1 = y1*2;
 8   } else {
 9     r = r*x;
10     y1 = y1+1;
11   }
12 }
```

Here, y_1*2 can be replaced by t

Available expressions

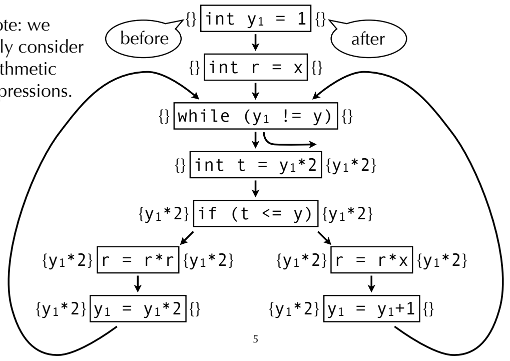
Why is the previous optimization valid?

Because at line 7, where expression y_1*2 appears for the second time, it is **available**. That is, no matter how we reach line 7, y_1*2 will have been computed previously at line 4. The computation of line 4 is still valid at line 7 because no redefinition of y_1 appears between those two points.

Generally speaking, we can define for every program point the set of **available expressions**, which is the set of all non-trivial expressions whose value has already been computed at that point.

Available expressions

Note: we only consider arithmetic expressions.

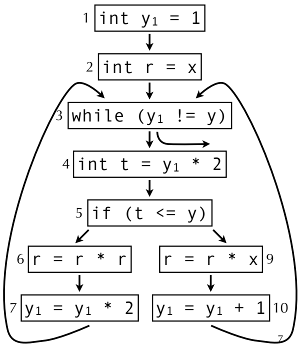


Formalizing the analysis

How can these ideas be formalized?

1. introduce a variable i_n for the set of expressions available before node *n*, and a variable o_n for the set of expressions available after node *n*,
2. define equations between those variables,
3. solve those equations.

Equations



$i_1=\{\}$	$o_1=i_1$
$i_2=o_1$	$o_2=i_2$
$i_3=o_2 \cap o_7 \cap o_{10}$	$o_3=i_3$
$i_4=o_3$	$o_4= \{\texttt{y}_1\texttt{*2}\} \cup i_4$
$i_5=o_4$	$o_5=i_5$
$i_6=o_5$	$o_6=i_6\downarrow\texttt{r}$
$i_7=o_6$	$o_7=i_7\downarrow\texttt{y}_1$
$i_9=o_5$	$o_9=i_9\downarrow\texttt{r}$
$i_{10}=o_9$	$o_{10}=i_{10}\downarrow\texttt{y}_1$

Notation:
$S\downarrow\texttt{x}$ =
$S\setminus$\{all expressions using x\}

Solving equations

The equations can be solved by iteration:

- initialize all sets $i_1, \ldots, i_{10}, o_1, \ldots, o_{10}$ to the set of all non-trivial expressions in the program, here $\{\texttt{y}_1\texttt{*2}, \texttt{y}_1\texttt{+1}, \texttt{r*r}, \texttt{r*x}\}$,
- viewing the equations as assignments, compute the “new” value of those sets,
- iterate until fixed point is reached.

Initialization is done that way because we are interested in finding the *largest* sets satisfying the equations: the larger a set is, the more information it conveys (for this analysis).

Solving equations

To simplify the equations, we can first replace all i_k variables by their value, to obtain a simpler system, and then solve that system.

For our example, we get:

$o_1 = \{\}$	$o_6 = o_5\downarrow\texttt{r}$
$o_2 = o_1$	$o_7 = o_6\downarrow\texttt{y}_1$
$o_3 = o_2 \cap o_7 \cap o_{10}$	$o_9 = o_5\downarrow\texttt{r}$
$o_4 = o_3 \cup \{\texttt{y}_1\texttt{*2}\}$	$o_{10} = o_9\downarrow\texttt{y}_1$
$o_5 = o_4$	

Solving equations

The simpler system can be solved by iterating until a fixed point is reached, which happens after 7 iterations.

It.	**1**	**2**	**3**	**4**	**5**	**6**	**7**
o_1	*YR*	{}	{}	{}	{}	{}	{}
o_2	*YR*	*YR*	{}	{}	{}	{}	{}
o_3	*YR*	*YR*	*R*	{}	{}	{}	{}
o_4	*YR*	*YR*	*YR*	{y₁*2, r*r, r*x}	{y₁*2}	{y₁*2}	{y₁*2}
o_5	*YR*	*YR*	*YR*	*YR*	{y₁*2, r*r, r*x}	{y₁*2}	{y₁*2}
o_6	*YR*	*Y*	*Y*	*Y*	*Y*	{y₁*2}	{y₁*2}
o_7	*YR*	*R*	{}	{}	{}	{}	{}
o_9	*YR*	*Y*	*Y*	*Y*	*Y*	{y₁*2}	{y₁*2}
o_{10}	*YR*	*R*	{}	{}	{}	{}	{}

Notation: $Y=\{\texttt{y}_1\texttt{*2}, \texttt{y}_1\texttt{+1}\}$, $R=\{\texttt{r*r}, \texttt{r*x}\}$, $YR = Y \cup R$

Generalization

In general, for a node n of the control-flow graph, the equations have the following form:

$i_n = o_{p1} \cap o_{p2} \cap \ldots \cap o_{pk}$

where $p_1 \ldots p_k$ are the predecessors of n.

$o_n = \text{gen}_{\text{AE}}(n) \cup (in \setminus \text{kill}_{\text{AE}}(n))$

where $\text{gen}_{\text{AE}}(n)$ are the non-trivial expressions computed by n, and $\text{kill}_{\text{AE}}(n)$ is the set of all non-trivial expressions that use a variable modified by n.

Substituting i_n in o_n, we obtain the following equation for o_n:

$o_n = \text{gen}_{\text{AE}}(n) \cup [(o_{p1} \cap o_{p2} \cap \ldots \cap o_{pk}) \setminus \text{kill}_{\text{AE}}(n)]$

These equations are the dataflow equations for the available expressions dataflow analysis.

Note: generated expressions

The equation giving the expressions available at the exit of node n is:

$o_n = \text{gen}_{\text{AE}}(n) \cup (i_n \setminus \text{kill}_{\text{AE}}(n))$

where $\text{gen}_{\text{AE}}(n)$ are the non-trivial expressions computed by n, and $\text{kill}_{\text{AE}}(n)$ is the set of all non-trivial expressions that use a variable modified by n.

In order for this equation to be correct, expressions that are computed by n but which use a variable modified by n must not be part of $\text{gen}_{\text{AE}}(n)$. For example

$\text{gen}_{\text{AE}}(\boxed{\texttt{x=y*y}}) = \{\texttt{y*y}\}$ but $\text{gen}_{\text{AE}}(\boxed{\texttt{y=y*y}}) = \{\}$

Dataflow analysis

Available expressions is one example of a dataflow analysis.

Dataflow analysis is a global analysis framework that can be used to approximate various properties of programs.

The results of those analyses can be used to perform several optimisations, for example:

- common sub-expression elimination, as we have seen,
- dead-code elimination,
- constant propagation,
- register allocation,
- etc.

Analysis scope

In this course, we will only consider intra-procedural dataflow analyses. That is, analyses that work on a single function at a time.

As in our example, those analyses work on the code of a function represented as a **control-flow graph** (**CFG**).

The nodes of the CFG are the statements of the function.

The edges of the CFG represent the flow of control: there is an edge from n_1 to n_2 if and only if control can flow immediately from n_1 to n_2. That is, if the statements of n_1 and n_2 can be executed in direct succession.

Analysis #2 Live variables

Live variable

A variable is said to be **live** at a given point if its value will be read later. While liveness is clearly undecidable, a conservative approximation can be computed using dataflow analysis.

This approximation can then be used, for example, to allocate registers: a set of variables that are never live at the same time can share a single register.

Intuitions

Intuitively, a variable is live after a node if it is live before any of its successors.

Moreover, a variable is live before node *n* if it is either read by *n*, or live after *n* and not written by *n*.

Finally, no variable is live after an exit node.

Equations

We associate to every node n a pair of variables (i_n, o_n) that give the set of variables live when the node is entered or exited, respectively. These variables are defined as follows:

$i_n = \mathrm{gen}_{LV}(n) \cup (o_n \setminus \mathrm{kill}_{LV}(n))$

where $\mathrm{gen}_{LV}(n)$ is the set of variables read by n, and $\mathrm{kill}_{LV}(n)$ is the set of variables written by n.

$o_n = i_{s1} \cup i_{s2} \cup \ldots \cup i_{sk}$

where $s_1 \ldots s_k$ are the successors of n.

Substituting o_n in i_n, we obtain the following equation for i_n:

$i_n = \mathrm{gen}_{LV}(n) \cup [(i_{s1} \cup i_{s2} \cup \ldots \cup i_{sk}) \setminus \mathrm{kill}_{LV}(n)]$

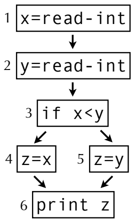
Equation solving

We are interested in finding the smallest sets of variables live at a given point, as the information conveyed by a set decreases as its size increases.

Therefore, to solve the equations by iteration, we initialize all sets with the empty set.

Example

CFG

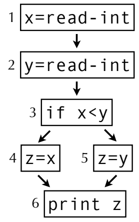


equations	**solution**
$i_1 = i_2 \setminus \{x\}$	$i_1 = \{\}$
$i_2 = i_3 \setminus \{y\}$	$i_2 = \{x\}$
$i_3 = \{x, y\} \cup (i_4 \cup i_5)$	$i_3 = \{x, y\}$
$i_4 = \{x\} \cup (i_6 \setminus \{z\})$	$i_4 = \{x\}$
$i_5 = \{y\} \cup (i_6 \setminus \{z\})$	$i_5 = \{y\}$
$i_6 = \{z\}$	$i_6 = \{z\}$

Using *live variables*

The previous analysis shows that neither x nor y are live at the same time as z. Therefore, z can be replaced by x or y, thereby removing one assignment.

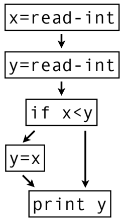
original CFG



analysis result

$i_1 = \{\}$
$i_2 = \{ x \}$
$i_3 = \{ x, y \}$
$i_4 = \{ x \}$
$i_5 = \{ y \}$
$i_6 = \{ z \}$

optimized CFG



Analysis #3 Reaching definitions

Reaching definitions

The **reaching definitions** for a program point are the assignments that may have defined the values of variables at that point.

Dataflow analysis can approximate the set of reaching definitions for all program points. These sets can then be used to perform constant propagation, for example.

Intuitions

Intuitively, a definition reaches the beginning of a node if it reaches the exit of any of its predecessors.

Moreover, a definition contained in a node *n* always reaches the end of *n* itself.

Finally, a definition reaches the end of a node *n* if it reaches the beginning of *n* and is not killed by *n* itself.

(A node *n* kills a definition *d* if and only if *n* is a definition and defines the same variable as *d*.)

As a first approximation, we consider that no definition reaches the beginning of the entry node.

Equations

We associate to every node n a pair of variables (i_n, o_n) that give the set of definitions reaching the entry and exit of n, respectively. These variables are defined as follows:

$i_n = o_{p1} \cup o_{p2} \cup \ldots \cup o_{pk}$

where $p_1 \ldots p_k$ are the predecessors of n.

$o_n = \text{gen}_{\text{RD}}(n) \cup (i_n \setminus \text{kill}_{\text{RD}}(n))$

where $\text{gen}_{\text{RD}}(n)$ is $\{n\}$ if n is a definition, $\{\}$ otherwise, and $\text{kill}_{\text{RD}}(n)$ is the set of definitions defining the same variable as n itself.

Substituting i_n in o_n, we obtain the following equation for o_n:

$o_n = \text{gen}_{\text{RD}}(n) \cup [(o_{p1} \cup o_{p2} \cup \ldots \cup o_{pk}) \setminus \text{kill}_{\text{RD}}(n)]$

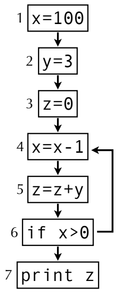
Equation solving

We are interested in finding the smallest sets of definitions reaching a point, as the information conveyed by a set decreases as its size increases.

Therefore, to solve the equations by iteration, we initialize all sets with the empty set.

Example

CFG



equations	**solution**
$o_1=\{(x,1)\}$	$o_1=\{(x,1)\}$
$o_2=\{(y,2)\} \cup o_1{\downarrow}y$	$o_2=\{(x,1), (y,2)\}$
$o_3=\{(z,3)\} \cup o_2{\downarrow}z$	$o_3=\{(x,1), (y,2), (z,3)\}$
$o_4=\{(x,4)\} \cup (o_3 \cup o_6){\downarrow}x$	$o_4=\{(x,4), (y,2), (z,3), (z,5)\}$
$o_5=\{(z,5)\} \cup o_4{\downarrow}z$	$o_5=\{(x,4), (y,2), (z,5)\}$
$o_6=o_5$	$o_6=\{(x,4), (y,2), (z,5)\}$
$o_7=o_6$	$o_7=\{(x,4), (y,2), (z,5)\}$

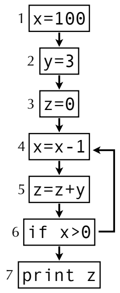
Notation:

$S{\downarrow}x = S \setminus$ {all definitions of x}

Using *reaching definitions*

The previous analysis shows that a single constant definition of y reaches node 5. Therefore, y can be replaced by 3 in node 5.

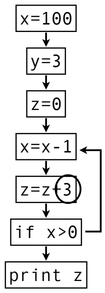
original CFG



analysis result

$o_1=\{(x,1)\}$
$o_2=\{(x,1), (y,2)\}$
$o_3=\{(x,1), (y,2), (z,3)\}$
$o_4=\{(x,4), (y,2), (z,3), (z,5)\}$
$o_5=\{(x,4), (y,2), (z,5)\}$
$o_6=\{(x,4), (y,2), (z,5)\}$
$o_7=\{(x,4), (y,2), (z,5)\}$

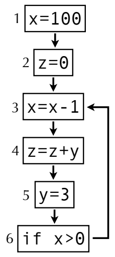
optimized CFG



Uninitialized variables

Note: if the language being analyzed permits uninitialized variables, the above analysis can produce incorrect results.

CFG



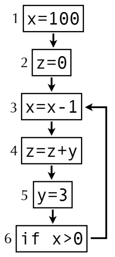
equations	**solution**
$o_1=\{(x,1)\}$	$o_1=\{(x,1)\}$
$o_2=\{(z,2)\} \cup o_1{\downarrow}z$	$o_2=\{(x,1), (z,2)\}$
$o_3=\{(x,3)\} \cup (o_2 \cup o_6){\downarrow}x$	$o_3=\{(x,3), (y,5), (z,2), (z,4)\}$
$o_4=\{(z,4)\} \cup o_3{\downarrow}z$	$o_4=\{(x,3), (y,5), (z,4)\}$
$o_5=\{(y,5)\} \cup o_4{\downarrow}y$	$o_5=\{(x,3), (y,5), (z,4)\}$
$o_6=o_5$	$o_6=\{(x,3), (y,5), (z,4)\}$

The solution can make us believe that y can safely be replaced by the value 3 in node 4, as before, but this is clearly wrong!

Uninitialized variables

If the language being analyzed permits uninitialised variables, all variables should be recorded as "initialized in some unknown location" at the entry of the first node!

CFG



equations	**solution**
$o_1=\{(x,1), (y,?), (z,?)\}$	$o_1=\{(x,1), (y,?), (z,?)\}$
$o_2=\{(z,2)\} \cup o_1{\downarrow}z$	$o_2=\{(x,1), (y,?), (z,2)\}$
$o_3=\{(x,3)\} \cup (o_2 \cup o_6){\downarrow}x$	$o_3=\{(x,3), (y,?), (y,5), (z,2), (z,4)\}$
$o_4=\{(z,4)\} \cup o_3{\downarrow}z$	$o_4=\{(x,3), (y,?), (y,5), (z,4)\}$
$o_5=\{(y,5)\} \cup o_4{\downarrow}y$	$o_5=\{(x,3), (y,5), (z,4)\}$
$o_6=o_5$	$o_6=\{(x,3), (y,5), (z,4)\}$

Analysis #4
Very busy expressions

Very busy expression

An expression is **very busy** at some program point if it will definitely be evaluated before its value changes.

Dataflow analysis can approximate the set of very busy expressions for all program points. The result of that analysis can then be used to perform code hoisting: the computation of a very busy expression can be performed at the earliest point where it is busy.

Intuitions

Intuitively, an expression is very busy after a node if it is very busy in all of its successors.

Moreover, an expression is very busy before node *n* if it is either evaluated by *n* itself, or very busy after *n* and not killed by *n*.

(A node kills an expression *e* if and only if it redefines a variable appearing in *e*.)

Finally, no expression is very busy after an exit node.

Equations

We associate to every node *n* a pair of variables (i_n,o_n) that give the set of expressions that are very busy when the node is entered or exited, respectively. These variables are defined as follows:

$i_n = \mathrm{gen}_{\mathrm{VB}}(n) \cup (o_n \setminus \mathrm{kill}_{\mathrm{VB}}(n))$

where $\mathrm{gen}_{\mathrm{VB}}(n)$ is the set of expressions evaluated by *n*, and $\mathrm{kill}_{\mathrm{VB}}(n)$ is the set of expressions killed by *n*,

$o_n = i_{s1} \cap i_{s2} \cap \ldots \cap i_{sk}$

where $s_1 \ldots s_k$ are the successors of *n*.

Substituting o_n in i_n, we obtain the following equation for i_n:

$i_n = \mathrm{gen}_{\mathrm{VB}}(n) \cup [(i_{s1} \cap i_{s2} \cap \ldots \cap i_{sk}) \setminus \mathrm{kill}_{\mathrm{VB}}(n)]$

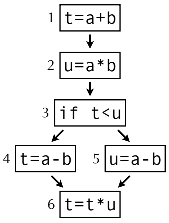
Equation solving

We are interested in finding the largest sets of very busy expressions, as the information conveyed by a set increases with its size.

Therefore, to solve the equations by iteration, we initialize all sets with the set of all non-trivial expressions appearing in the program.

Example

CFG

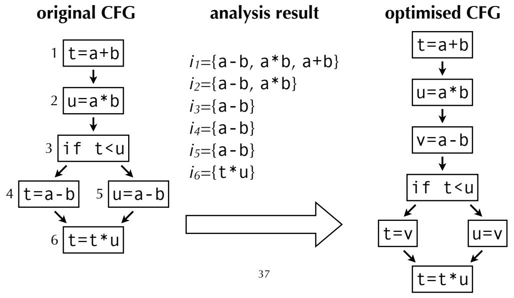


equations	**solution**
i_1={a+b} ∪ i_2↓t	i_1={a+b, a-b, a*b}
i_2={a*b} ∪ i_3↓u	i_2={a-b, a*b}
i_3=i_4 ∩ i_5	i_3={a-b}
i_4={a-b} ∪ i_6↓t	i_4={a-b}
i_5={a-b} ∪ i_6↓u	i_5={a-b}
i_6={t*u}	i_6={t*u}

Notation:
S↓x = *S* \ {all expressions using x}

Using *very busy expressions*

The previous analysis shows that `a-b` is very busy before the conditional. It can therefore be evaluated earlier.



original CFG **analysis result** **optimised CFG**

i_1={a-b, a*b, a+b}
i_2={a-b, a*b}
i_3={a-b}
i_4={a-b}
i_5={a-b}
i_6={t*u}

Classification of dataflow analyses

Equations summary

Analysis	Input equation	Output equation
available expressions	i_n = $o_{p1} \cap o_{p2} \cap \ldots \cap o_{pk}$	o_n = $\mathrm{gen}_{AE}(n) \cup (i_n \setminus \mathrm{kill}_{AE}(n))$
live variables	i_n = $\mathrm{gen}_{LV}(n) \cup (o_n \setminus \mathrm{kill}_{LV}(n))$	o_n = $i_{s1} \cup i_{s2} \cup \ldots \cup i_{sk}$
reaching definitions	i_n = $o_{p1} \cup o_{p2} \cup \ldots \cup o_{pk}$	o_n = $\mathrm{gen}_{RD}(n) \cup (i_n \setminus \mathrm{kill}_{RD}(n))$
very busy expressions	i_n = $\mathrm{gen}_{VB}(n) \cup (o_n \setminus \mathrm{kill}_{VB}(n))$	o_n = $i_{s1} \cap i_{s2} \cap \ldots \cap i_{sk}$

Taxonomy

Analyses for which the property of a node depends on those of its predecessors – *e.g.* available expressions, reaching definitions – are called **forward analyses**.

Analyses for which the property of a node depends on those of its successors – *e.g.* very busy expressions, live variables – are called **backward analyses**.

Analyses for which a property must be true in all successors or predecessors of a node to be true in that node – *e.g.* available, very busy expressions – are called **must analyses**.

Analyses for which a property must be true in at least one successor or predecessor of a node to be true in that node – *e.g.* reaching definitions, live variables – are called **may analyses**.

Speeding-up dataflow analyses

Speeding-up analyses

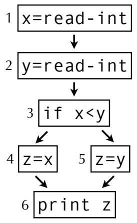
Several techniques can be used to speed up the various dataflow analyses:

- an algorithm based on a work-list can avoid useless computations,
- the equations can be ordered in order to propagate information faster,
- the analyses can be performed on smaller control-flow graphs, where nodes are basic blocks instead of individual instructions,
- bit-vectors can be used to represent sets.

Running example

We will reuse the live variable analysis example to illustrate the techniques used to speed up dataflow analyses.

CFG



equations	**solution**
$i_1 = i_2 \setminus \{x\}$	$i_1 = \{\}$
$i_2 = i_3 \setminus \{y\}$	$i_2 = \{x\}$
$i_3 = \{x, y\} \cup (i_4 \cup i_5)$	$i_3 = \{x, y\}$
$i_4 = \{x\} \cup (i_6 \setminus \{z\})$	$i_4 = \{x\}$
$i_5 = \{y\} \cup (i_6 \setminus \{z\})$	$i_5 = \{y\}$
$i_6 = \{z\}$	$i_6 = \{z\}$

Base case: iteration

Computing the solution to the equations using the standard iterative technique requires 3 iterations, each of which requires 6 computations, for a total of 18 computations:

Iteration	i_1	i_2	i_3	i_4	i_5	i_6
0	{ }	{ }	{ }	{ }	{ }	{ }
1	{ }	{ }	{ x, y }	{ x }	{ y }	{ z }
2	{ }	{ x }	{ x, y }	{ x }	{ y }	{ z }
3	{ }	{ x }	{ x, y }	{ x }	{ y }	{ z }

$i_1 = i_2\setminus\{x\}$, $i_2 = i_3\setminus\{y\}$, $i_3 = \{x,y\} \cup (i_4 \cup i_5)$,
$i_4 = \{x\} \cup (i_6\setminus\{z\})$, $i_5 = \{y\} \cup (i_6\setminus\{z\})$, $i_6 = \{z\}$

Work-list algorithm

Computing the fixed point by simple iteration as we did works, but is wasteful as the information for all nodes is re-computed at every iteration.

It is possible to do better by remembering, for every variable *v*, the set dep(*v*) of the variables whose value depends on the value of *v* itself.

Then, whenever the value of some variable *v* changes, we only re-compute the value of the variables that belong to dep(*v*).

Work-list algorithm in Scala

```
def solve[T](Eqs: Array[(Int => T) => T],
             dep: Int => List[Int],
             init: T): (Int => T) = {
  def loop(q: List[Int], sol: Map[Int,T]): (Int => T) =
    q match {
      case i :: is =>
        val y = Eqs(i)(sol)
        if (y == sol(i))
          loop(is, sol)
        else
          loop(is ::: (dep(i) diff q), sol + i->y)
      case Nil =>
        sol
    }
  loop(List.range(0, Eqs.length),
       Map.empty withDefaultValue init)
```

Work-list

It.	***q***	i_1	i_2	i_3	i_4	i_5	i_6
0	$[i_1,i_2,i_3,i_4,i_5,i_6]$	{}	{}	{}	{}	{}	{}
1	$[i_2,i_3,i_4,i_5,i_6]$	{}	{}	{}	{}	{}	{}
2	$[i_3,i_4,i_5,i_6]$	{}	{}	{}	{}	{}	{}
3	$[i_4,i_5,i_6,i_2]$	{}	{}	{x,y}	{}	{}	{}
4	$[i_5,i_6,i_2,i_3]$	{}	{}	{x,y}	{x}	{}	{}
5	$[i_6,i_2,i_3]$	{}	{}	{x,y}	{x}	{y}	{}
6	$[i_2,i_3,i_4,i_5]$	{}	{}	{x,y}	{x}	{y}	{z}
7	$[i_3,i_4,i_5,i_1]$	{}	{x}	{x,y}	{x}	{y}	{z}
8	$[i_4,i_5,i_1]$	{}	{x}	{x,y}	{x}	{y}	{z}
9	$[i_5,i_1]$	{}	{x}	{x,y}	{x}	{y}	{z}
10	$[i_1]$	{}	{x}	{x,y}	{x}	{y}	{z}
11	[]	{}	{x}	{x,y}	{x}	{y}	{z}

$i_1 = i_2\setminus\{x\}$, $i_2 = i_3\setminus\{y\}$, $i_3 = \{x,y\} \cup (i_4 \cup i_5)$,
$i_4 = \{x\} \cup (i_6\setminus\{z\})$, $i_5 = \{y\} \cup (i_6\setminus\{z\})$, $i_6 = \{z\}$

Node ordering

Using the work-list, "only" 11 computations were required to compute the result.

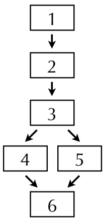
It is however clear that the process could be even faster if the elements of the work-list were ordered in the reverse order. This is because live variables analysis is a backward analysis.

The goal of node ordering is to order the elements of the work-list in such a way that the solution is computed as fast as possible.

(Reverse) post-order

For backward analyses, ordering the variables in the work-list according to a post-order traversal of the CFG nodes speeds up convergence. For forward analyses, reverse post-order has the same characteristic.

CFG



Post-order:
6 5 4 3 2 1 or 6 **4 5** 3 2 1
Reverse post-order:
1 2 3 4 5 6 or 1 2 3 **5 4** 6

Note: reverse post-order is not the same as pre-order!
Pre-order:
1 2 3 4 **6 5** or 1 2 3 **5 6 4**

Work-list with node ordering

By ordering the nodes in post-order, only 6 computations are required to obtain the solution.

It.	***q***	i_1	i_2	i_3	i_4	i_5	i_6
0	$[i_6,i_5,i_4,i_3,i_2,i_1]$	{}	{}	{}	{}	{}	{}
1	$[i_5,i_4,i_3,i_2,i_1]$	{}	{}	{}	{}	{}	{z}
2	$[i_4,i_3,i_2,i_1]$	{}	{}	{}	{}	{y}	{z}
3	$[i_3,i_2,i_1]$	{}	{}	{}	{x}	{y}	{z}
4	$[i_2,i_1]$	{}	{}	{x,y}	{x}	{y}	{z}
5	$[i_1]$	{}	{x}	{x,y}	{x}	{y}	{z}
6	[]	{}	{x}	{x,y}	{x}	{y}	{z}

$i_1 = i_2\backslash\{x\}$, $i_2 = i_3\backslash\{y\}$, $i_3 = \{x,y\} \cup (i_4 \cup i_5)$,
$i_4 = \{x\} \cup (i_6\backslash\{z\})$, $i_5 = \{y\} \cup (i_6\backslash\{z\})$, $i_6 = \{z\}$

Working with basic blocks

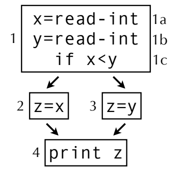
Until now, CFG nodes were single instructions. In practice, **basic blocks** tend to be used as nodes, to reduce the size of the CFG.

When dataflow analysis is performed on a CFG composed of basic blocks, a pair of variables is attached to every block, not to every instruction.

Once the solution is known for basic blocks, computing the solution for individual instructions is easy.

CFG with basic blocks

CFG



```
  x=read-int  1a
1 y=read-int  1b
   if x<y     1c

2 z=x    3 z=y

  4 print z
```

equations

$i_1 = (i_2 \cup i_3) \setminus \{x, y\}$
$i_2 = \{x\} \cup (i_4 \setminus \{z\})$
$i_3 = \{y\} \cup (i_4 \setminus \{z\})$
$i_4 = \{z\}$

solution

$i_1 = \{\}$
$i_2 = \{x\}$
$i_3 = \{y\}$
$i_4 = \{z\}$

The solution for individual instructions is computed from the basic-block solution, in a single pass – here backwards:

$i_{1c} = \{x, y\} \cup (i_2 \cup i_3) = \{x, y\}$
$i_{1b} = i_{1c} \setminus \{y\} = \{x\}$
$i_{1a} = i_{1b} \setminus \{x\} = \{\}$

Bit vectors

All dataflow analyses we have seen work on sets of values.

If these sets are dense, a good way to represent them is to use bit vectors: a bit is associated to every possible element of the set, and its value is 1 if and only if the corresponding element belongs to the set.

On such a representation, set union is bitwise-or, set intersection is bitwise-and, set difference is bitwise-and composed with bitwise-negation.

Bit vector example

original equations

$i_1 = i_2 \setminus \{x\}$
$i_2 = i_3 \setminus \{y\}$
$i_3 = \{x, y\} \cup (i_4 \cup i_5)$
$i_4 = \{x\} \cup (i_6 \setminus \{z\})$
$i_5 = \{y\} \cup (i_6 \setminus \{z\})$
$i_6 = \{z\}$

original solution

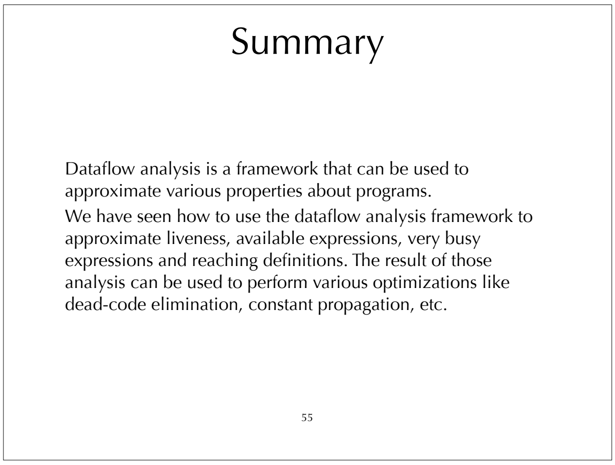
$i_1 = \{\}$
$i_2 = \{x\}$
$i_3 = \{x, y\}$
$i_4 = \{x\}$
$i_5 = \{y\}$
$i_6 = \{z\}$

bit vector equations

$i_1 = i_2\ \&\ \sim 100$
$i_2 = i_3\ \&\ \sim 010$
$i_3 = 110 \mid (i_4 \mid i_5)$
$i_4 = 100 \mid (i_6\ \&\ \sim 001)$
$i_5 = 010 \mid (i_6\ \&\ \sim 001)$
$i_6 = 001$

bit vector solution

$i_1 = 000$
$i_2 = 100$
$i_3 = 110$
$i_4 = 100$
$i_5 = 010$
$i_6 = 001$



Summary

Dataflow analysis is a framework that can be used to approximate various properties about programs.

We have seen how to use the dataflow analysis framework to approximate liveness, available expressions, very busy expressions and reaching definitions. The result of those analysis can be used to perform various optimizations like dead-code elimination, constant propagation, etc.