Dataflow analysis

Advanced Compiler Construction
Michel Schinz – 2020-03-19

A first example : available expressions

CSE

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 optimization valid?

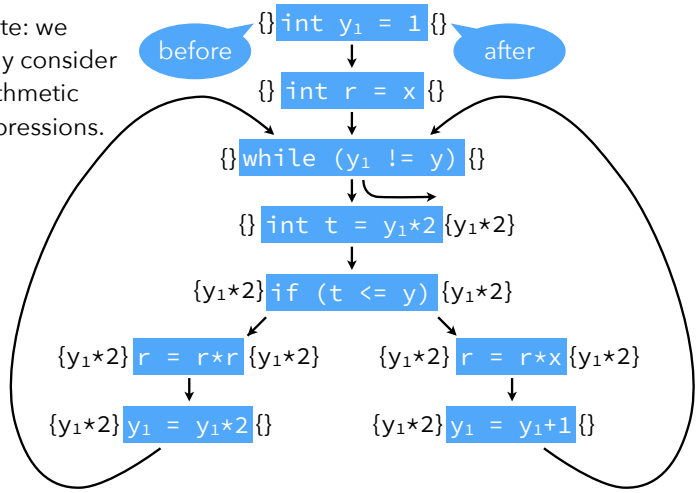
Because at line 7, expression y_1*2 is **available** :

- no matter how we reach line 7, y_1*2 will have been computed previously at line 4,
- that computation is still valid at line 7 (no redefinition of y_1 between those two points).

We can define for every program point the set of **available expressions** :
the set of all non-trivial expressions whose value has already been computed at that point.

Available expressions

Note: we only consider arithmetic expressions.

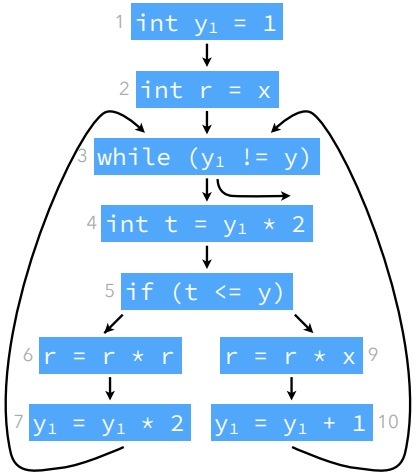


Formalizing the analysis

How can these ideas be formalized?

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

Equations



$i_1 = \{\}$ $o_1 = i_1$

$i_2 = o_1$ $o_2 = i_2$

$i_3 = o_2 \cap o_7 \cap o_{10}$ $o_3 = i_3$

$i_4 = o_3$ $o_4 = \{y_1*2\} \cup i_4$

$i_5 = o_4$ $o_5 = i_5$

$i_6 = o_5$ $o_6 = i_6 \downarrow r$

$i_7 = o_6$ $o_7 = i_7 \downarrow y_1$

$i_9 = o_5$ $o_9 = i_9 \downarrow r$

$i_{10} = o_9$ $o_{10} = i_{10} \downarrow y_1$

Notation:

$S \downarrow x =$
$S \setminus \{\text{all expressions using } x\}$

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 $\{y_1*2, y_1+1, r*r, r*x\}$,
- 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

Simplify by replacing all i_k variables by their value:

$o_1 = \{\}$
$o_2 = o_1$
$o_3 = o_2 \cap o_7 \cap o_{10}$
$o_4 = o_3 \cup \{y_1 \star 2\}$
$o_5 = o_4$
$o_6 = o_5 \downarrow r$
$o_7 = o_6 \downarrow y_1$
$o_9 = o_5 \downarrow r$
$o_{10} = o_9 \downarrow y_1$

Solving equations

The simplified system is solved 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*	$\{y_1\star2, r\star r, r\star x\}$	$\{y_1\star2\}$	$\{y_1\star2\}$	$\{y_1\star2\}$
o_5	*YR*	*YR*	*YR*	*YR*	$\{y_1\star2, r\star r, r\star x\}$	$\{y_1\star2\}$	$\{y_1\star2\}$
o_6	*YR*	*Y*	*Y*	*Y*	*Y*	$\{y_1\star2\}$	$\{y_1\star2\}$
o_7	*YR*	*R*	{}	{}	{}	{}	{}
o_9	*YR*	*Y*	*Y*	*Y*	*Y*	$\{y_1\star2\}$	$\{y_1\star2\}$
o_{10}	*YR*	*R*	{}	{}	{}	{}	{}

Notation: $Y = \{y_1\star2, y_1+1\}$, $R = \{r\star r, r\star x\}$, $YR = Y \cup R$

Generalization

The equations for a node n 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.

$o_n = gen_{AE}(n) \cup (i_n \setminus kill_{AE}(n))$
where $gen_{AE}(n)$ are the non-trivial expressions computed by n, and $kill_{AE}(n)$ is the set of all non-trivial expressions that use a variable modified by n.

Substituting i_n in o_n, we get:

$o_n = gen_{AE}(n) \cup [(o_{p1} \cap o_{p2} \cap \ldots \cap o_{pk}) \setminus kill_{AE}(n)]$

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

Generated expressions

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

$o_n = gen_{AE}(n) \cup (i_n \setminus kill_{AE}(n))$

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

For this to be correct, expressions that are computed by n but that use a variable modified by n must not be part of $gen_{AE}(n)$. For example

$gen_{AE}(x=y\star y) = \{y\star y\}$ but $gen_{AE}(y=y\star y) = \{\}$

Dataflow analysis

Available expressions is one example of a **dataflow analysis**.
Dataflow analysis is an analysis framework that can approximate various properties of programs, which can then be used to do:

- common sub-expression elimination,
- dead code elimination,
- constant propagation,
- register allocation,
- etc.

Analysis scope

We only consider **intra-procedural dataflow analyses** (i.e. restricted to a single function) working on the control-flow graph of the function.
Reminder:

- CFG nodes are the statements of the function,
- CFG edges represent the flow of control: an edge from n_1 to n_2 means that control can flow immediately from n_1 to n_2.

Analysis #2: live variables

Live variable

A variable is said to be **live** at a given point if its value will be read later:

- clearly undecidable, but
- dataflow analysis can compute an approximation.

Liveness can be used to allocate registers, for example: 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,
- a variable is live before node n if it is either read by n, or live after n and not written by n,
- 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 that are live when the node is entered or exited, respectively, defined as follows:

$i_n = gen_{LV}(n) \cup (o_n \setminus kill_{LV}(n))$

where $gen_{LV}(n)$ is the set of variables read by n, and $kill_{LV}(n)$ 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 get:

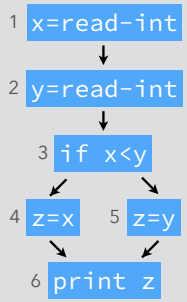
$i_n = gen_{LV}(n) \cup [(i_{s1} \cup i_{s2} \cup \ldots \cup i_{sk}) \setminus kill_{LV}(n)]$

Equation solving

We want the sets of live variables to be as small as possible.
Therefore we initialize all sets to the empty set.

Example

CFG



equations

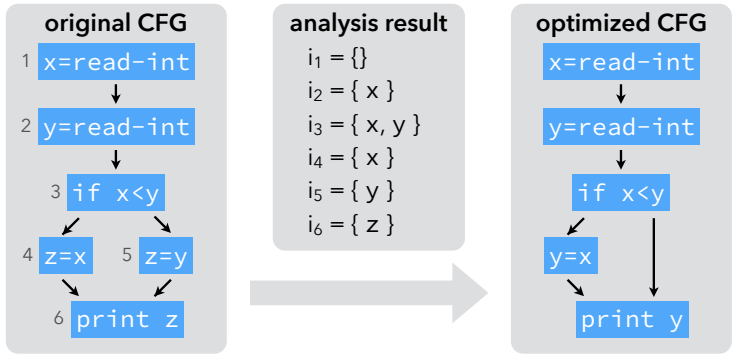
$i_1 = i_2 \setminus \{x\}$
$i_2 = i_3 \setminus \{y\}$
$i_3 = \{x, y\} \cup (i_4 \cup i_5)$
$i_4 = \{x\} \cup (i_6 \setminus \{z\})$
$i_5 = \{y\} \cup (i_6 \setminus \{z\})$
$i_6 = \{z\}$

solution

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

Using *live variables*

Neither x nor y is live at the same time as z.
Therefore, z can be replaced by x or y.



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.
Can be approximated using dataflow analysis, and the result 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,
- a definition contained in a node n always reaches the end of n itself,
- 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.)
For now, we assume 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, 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.

$o_n = gen_{RD}(n) \cup (i_n \setminus kill_{RD}(n))$

where $gen_{RD}(n)$ is {(x,n)} if n is a definition of variable x, {} otherwise, and $kill_{RD}(n)$ is the set of definitions defining the same variable as n itself.

Substituting i_n in o_n, we get:

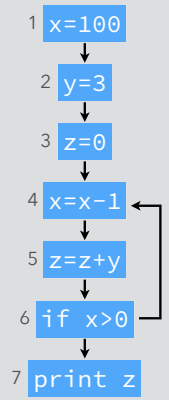
$o_n = gen_{RD}(n) \cup [(o_{p1} \cup o_{p2} \cup \ldots \cup o_{pk}) \setminus kill_{RD}(n)]$

Equation solving

We want the sets of reaching definitions to be as small as possible.
Therefore we initialize all sets to the empty set.

Example

CFG



equations

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

solution

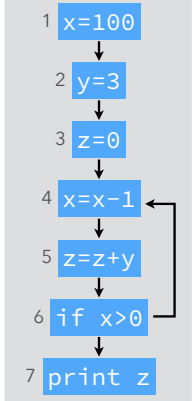
$o_1 = \{(x,1)\}$
$o_2 = \{(x,1), (y,2)\}$
$o_3 = \{(x,1), (y,2), (z,3)\}$
$o_4 = \{(x,4), (y,2), (z,3), (z,5)\}$
$o_5 = \{(x,4), (y,2), (z,5)\}$
$o_6 = \{(x,4), (y,2), (z,5)\}$
$o_7 = \{(x,4), (y,2), (z,5)\}$

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

Using *reaching definitions*

A single constant definition of y reaches node 5.
Therefore, y can be replaced by 3 in it.

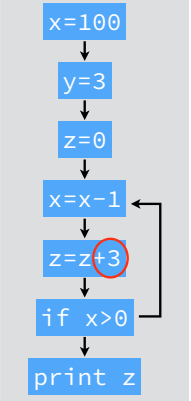
original CFG



analysis result

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

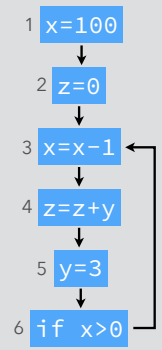
optimized



Uninitialized variables

Note: if uninitialized variables are allowed, the above analysis can produce incorrect results.

CFG



equations

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

solution

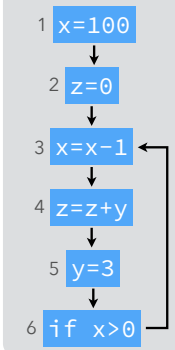
$o_1=\{(x,1)\}$
$o_2=\{(x,1), (z,2)\}$
$o_3=\{(x,3), (y,5), (z,2), (z,4)\}$
$o_4=\{(x,3), (y,5), (z,4)\}$
$o_5=\{(x,3), (y,5), (z,4)\}$
$o_6=\{(x,3), (y,5), (z,4)\}$

We might think that y can be replaced by the value 3 in node 4, as before, but it's wrong!

Uninitialized variables

Solution: record all variables as "initialized in some unknown location" at the entry of the first node!

CFG



equations

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

solution

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

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.
Can be approximated by dataflow analysis, and the result 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,
- 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,
- no expression is very busy after an exit node.

(A node kills an expression e iff it redefines a variable appearing in e.)

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, defined as follows:

$i_n = gen_{VB}(n) \cup (o_n \setminus kill_{VB}(n))$

where $gen_{VB}(n)$ is the set of expressions evaluated by n, and $kill_{VB}(n)$ 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 get:

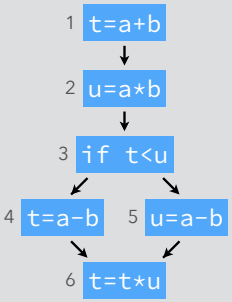
$i_n = gen_{VB}(n) \cup [(i_{s1} \cap i_{s2} \cap \ldots \cap i_{sk}) \setminus kill_{VB}(n)]$

Equation solving

We want the sets of very busy expressions to be as large as possible. Therefore we initialize all sets to the set of all non-trivial expressions of the function.

Example

CFG



equations

$i_1 = \{a+b\} \cup i_2 \downarrow t$
$i_2 = \{a*b\} \cup i_3 \downarrow u$
$i_3 = i_4 \cap i_5$
$i_4 = \{a-b\} \cup i_6 \downarrow t$
$i_5 = \{a-b\} \cup i_6 \downarrow u$
$i_6 = \{t*u\}$

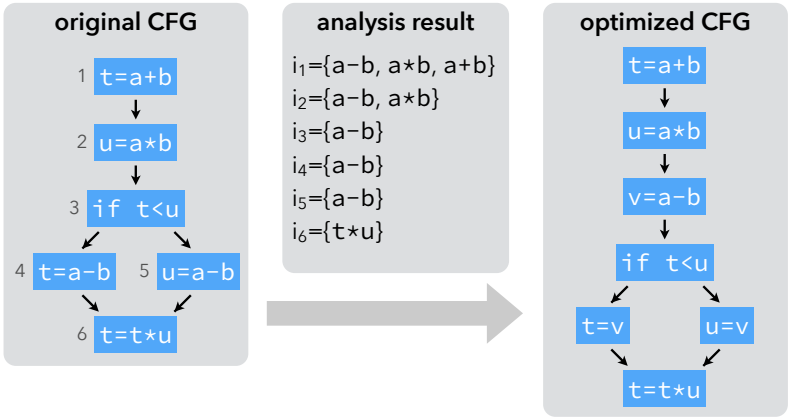
solution

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

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

Using *very busy expressions*

Expression `a-b` is very busy before the conditional.
Therefore, it can be evaluated earlier.



Classification of dataflow analyses

Equations summary

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

Taxonomy

Forward vs backward:

- **Forward analyses**: the property of a node depends on those of its predecessors.
- **Backward analyses**: the property of a node depends on those if its successors.

May vs must:

- **Must analyses**: a property must be true in all neighbors to be true in a node.
- **May analyses**: a property must be true in at least one neighbor to be true in a node.

Taxonomy

	Forward	Backward
Must	available expressions	very busy expressions
May	reaching definitions	live variables

Speeding-up dataflow analyses

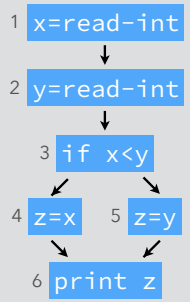
Speeding-up analyses

Dataflow analyses can be sped up by:
- a work-list algorithm, avoiding useless computations,
- equations ordering, speeding-up propagation,
- smaller CFGs using basic blocks,
- bit-vectors to represent sets.

Running example

To illustrate speed-up techniques, we reuse the live variables example:

CFG



equations

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

solution

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

Base case: iteration

Computation by iteration: 3 iterations with 6 computations each, 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 }

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

Work-list algorithm

Work-list algorithm:

- remember, for every variable v, the set dep(v) of the variables whose value depends on v,
- whenever some variable v changes, only re-compute the variables that belong to dep(v).

Work-list algorithm in Scala

```
def solve[T](eqs: Seq[(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	[$\mathbf{i_1}$,i_2,i_3,i_4,i_5,i_6]	{}	{}	{}	{}	{}	{}
1	[$\mathbf{i_2}$,i_3,i_4,i_5,i_6]	{}	{}	{}	{}	{}	{}
2	[$\mathbf{i_3}$,i_4,i_5,i_6]	{}	{}	{}	{}	{}	{}
3	[$\mathbf{i_4}$,i_5,i_6,i_2]	{}	{}	{x,y}	{}	{}	{}
4	[$\mathbf{i_5}$,i_6,i_2,i_3]	{}	{}	{x,y}	{x}	{}	{}
5	[$\mathbf{i_6}$,i_2,i_3]	{}	{}	{x,y}	{x}	{y}	{}
6	[$\mathbf{i_2}$,i_3,i_4,i_5]	{}	{}	{x,y}	{x}	{y}	{z}
7	[$\mathbf{i_3}$,i_4,i_5,i_1]	{}	{x}	{x,y}	{x}	{y}	{z}
8	[$\mathbf{i_4}$,i_5,i_1]	{}	{x}	{x,y}	{x}	{y}	{z}
9	[$\mathbf{i_5}$,i_1]	{}	{x}	{x,y}	{x}	{y}	{z}
10	[$\mathbf{i_1}$]	{}	{x}	{x,y}	{x}	{y}	{z}
11	[]	{}	{x}	{x,y}	{x}	{y}	{z}

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

Node ordering

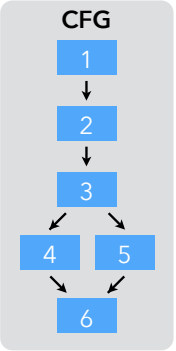
The work-list algorithm needs "only" 11 computations, but:
- would be faster with work-list reversed,
- that's because live variables is a backward analysis.

Node ordering orders the elements of the work-list so that the solution is computed as fast as possible.

(Reverse) post-order

Backward analyses: use post-order traversal of CFG.

Forward analyses: use reverse post-order traversal of 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**

Post-order work-list

Post-order work-list: only 6 computations required.

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

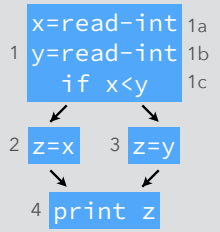
$i_1 = i_2 \setminus \{x\}$, $i_2 = i_3 \setminus \{y\}$, $i_3 = \{x,y\} \cup (i_4 \cup i_5)$,
$i_4 = \{x\} \cup (i_6 \setminus \{z\})$, $i_5 = \{y\} \cup (i_6 \setminus \{z\})$, $i_6 = \{z\}$

Basic blocks

CFG nodes can be **basic blocks** instead of instructions:
- reduces the size of the CFG,
- variables are attached to basic blocks, not instructions,
- computing the solution for instructions is easy given that for basic blocks.

CFG with basic blocks

CFG



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 they are dense, represent them as bit vectors:

- uses only one bit per element,
- union is "bitwise or",
- intersection is "bitwise and",
- complement is "bitwise inversion",
- etc.

Bit vectors 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 programs properties, e.g.:

- liveness,
- available expressions,
- very busy expressions,
- reaching definitions.

These approximations can be used for optimizations like:

- register allocation,
- constant propagation,
- etc.