# Data-flow analysis (II)

Michel Schinz
(based on material by Michael Schwartzbach and Erik Stenman)
Advanced Compiler Construction / 2006-06-09



# Data-flow analyses

We have already seen how to use the data-flow analysis framework to compute an approximation of liveness for each program point.

We will now explore other kinds of information about programs which can be computed using that framework.



# Data-flow analysis Available expressions



# Available expressions

A non-trivial expression in a program is **available** at some point if its value has already been computed earlier.

Data-flow analysis can be used to approximate the set of expressions available at all program points. The result from that analysis can then be used to eliminate common sub-expressions, for example.



# Available expressions intuitions

We will compute the set of expressions available *after* every node of the CFG.

Intuitively, an expression *e* is available after some node *n* if it is available after all predecessors of *n*, or if it is defined by *n* itself, and not killed by *n*.

A node *n* kills an expression *e* if it gives a new value to a variable used by *e*. For example, the assignment `x←y` kills all expressions which use `x`, like `x+1`.



# Available expressions equations

To approximate available expressions, we attach to every node *n* of the CFG a variable $v_n$ containing the set of expressions available *after* it.

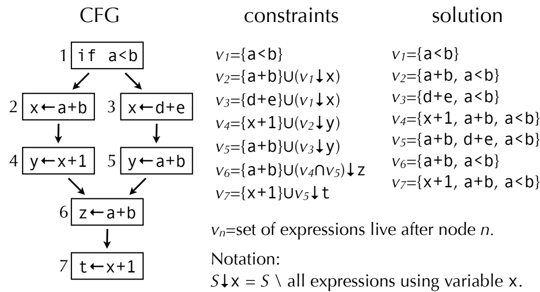
Then we derive constraints from the CFG nodes, which have the form:

$$v_n = (v_{p1} \cap v_{p2} \cap \ldots \setminus \text{kill}(n)) \cup \text{gen}(n)$$

where gen(*n*) is the set of expressions computed by *n*, and kill(*n*) the set of expressions killed by *n*.

# Available expressions example

CFG | constraints | solution




Constraints:

$v_1$={a<b}
$v_2$={a+b}∪($v_1$↓x)
$v_3$={d+e}∪($v_1$↓x)
$v_4$={x+1}∪($v_2$↓y)
$v_5$={a+b}∪($v_3$↓y)
$v_6$={a+b}∪($v_4$∩$v_5$)↓z
$v_7$={x+1}∪$v_5$↓t

Solution:

$v_1$={a<b}
$v_2$={a+b, a<b}
$v_3$={d+e, a<b}
$v_4$={x+1, a+b, a<b}
$v_5$={a+b, d+e, a<b}
$v_6$={a+b, a<b}
$v_7$={x+1, a+b, a<b}

$v_n$=set of expressions live after node $n$.

Notation:
$S$↓x = $S$ \ all expressions using variable x.



# Data-flow analysis Very busy expressions



# Very busy expressions

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

Data-flow 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 *e* can be performed at the earliest point where it is busy.



# Very busy expressions intuitions

We will compute the set of very busy expressions before every node of the CFG.

Intuitively, an expression *e* is very busy before node *n* if it is evaluated by *n*, or if it is very busy in all successors of *n*, and it is not killed by *n*.



# Very busy expressions equations

To approximate very busy expressions, we attach to node *n* of the CFG a variable $v_n$ containing the set of expressions which are very busy *before* it.

Then we derive constraints from the CFG nodes, which have the form:

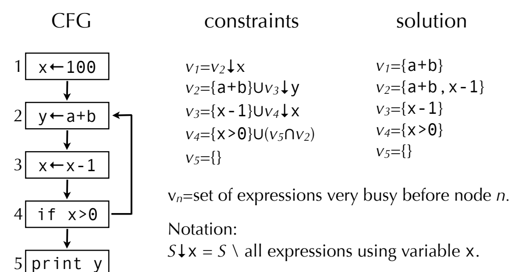
$$v_n = (v_{s1} \cap v_{s2} \cap \ldots \setminus \mathrm{kill}(n)) \cup \mathrm{gen}(n)$$

where gen($n$) is the set of expressions computed by $n$, and kill($n$) the set of expressions killed by $n$.



# Very busy expressions example

CFG | constraints | solution




Constraints:

$v_1$=$v_2$↓x
$v_2$={a+b}∪$v_3$↓y
$v_3$={x-1}∪$v_4$↓x
$v_4$={x>0}∪($v_5$∩$v_2$)
$v_5$={}

Solution:

$v_1$={a+b}
$v_2$={a+b,x-1}
$v_3$={x-1}
$v_4$={x>0}
$v_5$={}

$v_n$=set of expressions very busy before node $n$.

Notation:
$S$↓x = $S$ \ all expressions using variable x.

# Data-flow analysis 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.

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



# Reaching definitions intuitions

We will compute the set of reaching definitions after every node of the CFG. That set will be represented as a set of CFG node identifiers.

Intuitively, the reaching definitions after a node *n* are all the reaching definitions of the predecessors of *n*, minus those which define a variable defined by *n* itself, plus *n* itself.



# Reaching definitions equations (1)

To approximate reaching definitions, we attach to node *n* of the CFG a variable $v_n$ containing the set of definitions (CFG nodes) which can reach *n*.

For a node *n* which is not an assignment, the reaching definitions are simply those of its predecessors:

$$v_n = (v_{p1} \cup v_{p2} \cup \dots)$$



# Reaching definitions equations (2)

For a node *n* which is an assignment, the equation is more complicated:

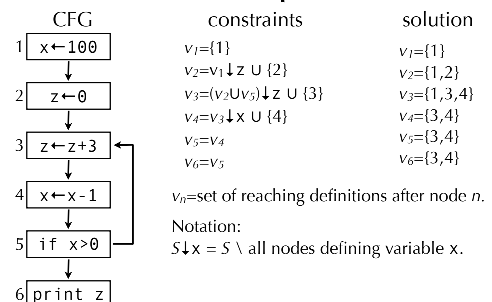
$$v_n = (v_{p1} \cup v_{p2} \cup \dots) \setminus \text{kill}(n) \cup \{ n \}$$

where kill(*n*) are the definitions killed by *n*, *i.e.* those which define the same variable as *n* itself.

For example, a definition like x←y kills all expressions of the form x←...



# Reaching definitions example




| constraints | solution |
| --- | --- |
| $v_1=\{1\}$ | $v_1=\{1\}$ |
| $v_2=v_1\downarrow z \cup \{2\}$ | $v_2=\{1,2\}$ |
| $v_3=(v_2 \cup v_5)\downarrow z \cup \{3\}$ | $v_3=\{1,3,4\}$ |
| $v_4=v_3\downarrow x \cup \{4\}$ | $v_4=\{3,4\}$ |
| $v_5=v_4$ | $v_5=\{3,4\}$ |
| $v_6=v_5$ | $v_6=\{3,4\}$ |

$v_n$=set of reaching definitions after node *n*.

Notation:
$S\downarrow x = S \setminus$ all nodes defining variable x.

# Using the result of data-flow analyses



# Using data-flow analysis

Once a particular data-flow analysis has been conducted, its result can be used to optimise the analysed program.

We will quickly examine some transformations which can be performed using the data-flow analysis presented before.



# Dead-code elimination

Useless assignments can be eliminated using liveness analysis, as follows:

> Whenever a CFG node *n* is of the form `x←e`, and `x` is not live after *n*, then the assignment is useless and node *n* can be removed.



# Common sub-expression elimination

Common sub-expressions can be eliminated using availability information, as follows:

> Whenever a CFG node *n* computes an expression of the form `x` *op* `y` and `x` *op* `y` is available before *n*, then the computation within *n* can be replaced by a reference to the previously-computed value.



# Constant propagation

Constant propagation can be performed using the result of reaching definitions analysis, as follows:

> When a CFG node *n* uses a value `x` and the only definition of `x` reaching *n* has the form `x←c` where `c` is a constant, then the use of `x` in *n* can be replaced by `c`.



# Copy propagation

Copy propagation – very similar to constant propagation – can be performed using the result of reaching definitions analysis, as follows:

> When a CFG node *n* uses a value `x`, and the only definition of `x` reaching *n* has the form `x←y` where `y` is a variable, and `y` is not redefined on any path leading to *n*, then the use of `x` in *n* can be replaced by `y`.

# Register allocation

To assign machine registers to program variables, liveness analysis is required.

We will explore register allocation in more details later, but intuitively it should be clear that a set of variables $V=\{ v_1, v_2, \ldots, v_n \}$ can be allocated to a single machine register provided that no two variables in $V$ are live simultaneously.



# Summary

Data-flow analysis is a framework that can be used to approximate various properties about programs.

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