Data-flow analysis

Michel Schinz – based on material by Erik Stenman and Michael Schwartzbach

Introduction to data-flow analysis

Data-flow analysis

Data-flow analysis is a global analysis framework that can be used to compute – or, more precisely, approximate – various properties of programs.

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

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

Example: liveness

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 data-flow 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.

Requirements

Data-flow analysis requires the program to be represented as a control flow graph (CFG).

To compute properties about the program, it assigns values to the nodes of the CFG. Those values must be related to each other by a special kind of partial order called a lattice.

We therefore start by introducing control flow graphs and lattice theory.

Control-flow graphs

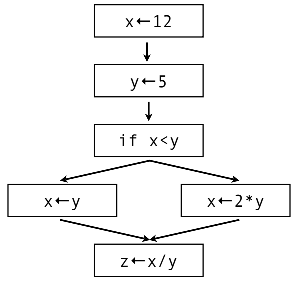
Control-flow graph

A **control flow graph** (**CFG**) is a graphical representation of a program.

The nodes of the CFG are the statements of that program.

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.

CFG example



Predecessors and successors

In the CFG, the set of the immediate predecessors of a node *n* is written pred(*n*).

Similarly, the set of the immediate successors of a node *n* is written succ(*n*).

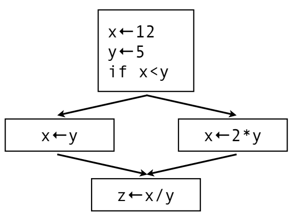
Basic block

A **basic block** is a maximal sequence of statements for which control flow is purely linear.

That is, control always enters a basic block from the top – its first instruction – and leaves from the bottom – its last instruction.

Basic blocks are often used as the nodes of a CFG, in order to reduce its size.

CFG example (basic blocks)



Lattice theory

Partial order

A **partial order** is a mathematical structure $(S,\sqsubseteq)$composed of a set S and a binary relation $\sqsubseteq$ on S, satisfying the following conditions:

1. reflexivity: $\forall x \in S,\ x \sqsubseteq x$
2. transitivity: $\forall x,y,z \in S,\ x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$
3. anti-symmetry: $\forall x,y \in S,\ x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$

Partial order example

In Java, the set of types along with the subtyping relation form a partial order.

According to that order, the type `String` is smaller (*i.e.* a subtype) of the type `Object`.

The type `String` and `Integer` are not comparable: none of them is a subtype of the other.

Upper bound

Given a partial order $(S,\sqsubseteq)$ and a set $X \subseteq S$, $y \in S$ is an **upper bound** for X, written $X \sqsubseteq y$, if

$\forall x \in X,\ x \sqsubseteq y$.

A **least upper bound** (**lub**) for X, written $\sqcup X$, is defined by:

$X \sqsubseteq \sqcup X \wedge \forall y \in S,\ X \sqsubseteq y \Rightarrow \sqcup X \sqsubseteq y$

Notice that a least upper bound does not always exist.

Lower bound

Given a partial order $(S, \sqsubseteq)$ and a set $X \subseteq S$, $y \in S$ is a **lower bound** for X, written $y \sqsubseteq X$, if

$\forall x \in X,\ y \sqsubseteq x$.

A **greatest lower bound** for X, written $\sqcap X$, is defined by:

$\sqcap X \sqsubseteq X \wedge \forall y \in S,\ y \sqsubseteq X \Rightarrow y \sqsubseteq \sqcap X$

Notice that a greatest lower bound does not always exist.

Lattice

A **lattice** is a partial order $L = (S,\sqsubseteq)$ for which $\sqcup X$ and $\sqcap X$ exist for all $X \subseteq S$.

A lattice has a unique greatest element, written $\top$ and pronounced "**top**", defined as $\top = \sqcup S$.

It also has a unique smallest element, written $\bot$ and pronounced "**bottom**", defined as $\bot = \sqcap S$.

The **height** of a lattice is the length of the longest path from $\bot$ to $\top$.

Finite partial orders

A partial order $(S,\sqsubseteq)$ is **finite** if the set S contains a finite number of elements.

For such partial orders, the lattice requirements reduce to the following:

- $\top$ and $\bot$ exist,
- every pair of elements x,y in S has a least upper bound – written $x \sqcup y$ – as well as a greatest lower bound – written $x \sqcap y$.

Cover relation

In a partial order $(S,\sqsubseteq)$, we say that an element y **covers** another element x if:

$(x \sqsubset y) \wedge (\forall z \in S, x \sqsubseteq z \sqsubset y \Rightarrow x = z)$

where $x \sqsubset y \Leftrightarrow x \sqsubseteq y \wedge x \neq y$.

Intuitively, y covers x if y is the smallest element greater than x.

Hasse diagram

A partial order can be represented graphically by a **Hasse diagram**.

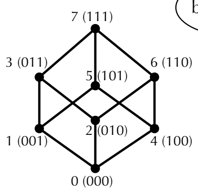
In such a diagram, the elements of the set are represented by dots.

If an element y covers an element x, then the dot of y is placed above the dot of x, and a line is drawn to connect the two dots.

Hasse diagram example

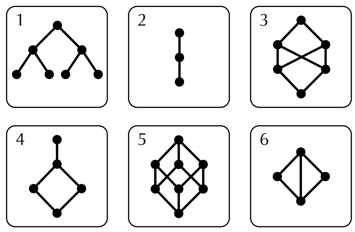
Hasse diagram for the partial order $(S,\sqsubseteq)$ where $S = \{ 0, 1, \ldots, 7 \}$ and $x \sqsubseteq y \Leftrightarrow (x \,\&\, y) = x$

bitwise and



Partial order examples

Which of the following partial orders are lattices?



Fixed points

Monotone function

A function $f : L \rightarrow L$ is **monotone** if and only if:

$\forall x,y \in S, x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$

This does *not* imply that f is increasing, as constant functions are also monotone.

Viewed as functions, $\sqcap$ and $\sqcup$ are monotone in both arguments.

Fixed point theorem

Definition: a value v is a **fixed point** of a function f if and only if $f(v) = v$.

Fixed point theorem: In a lattice L with finite height, every monotone function f has a unique least fixed point fix(f), and it is given by:

$\mathrm{fix}(f) = \bot \sqcup f(\bot) \sqcup f^2(\bot) \sqcup f^3(\bot) \sqcup \ldots$

Fixed points and equations

Fixed points are interesting as they enable us to solve systems of equations of the following form:

$x_1 = F_1(x_1, \ldots, x_n)$
$x_2 = F_2(x_1, \ldots, x_n)$
…
$x_n = F_n(x_1, \ldots, x_n)$

where $x_1, ..., x_n$ are variables, and $F_1, ..., F_n : L^n \to L$ are monotone functions.

Such a system has a unique least solution that is the least fixed point of the composite function $F : L^n \to L^n$ defined as:

$F(x_1, \ldots, x_n) = (F_1(x_1, \ldots, x_n), \ldots, F_n(x_1, \ldots, x_n))$

Fixed points and inequations

Systems of inequations of the following form:

$x_1 \sqsubseteq F_1(x_1, \ldots, x_n)$
$x_2 \sqsubseteq F_2(x_1, \ldots, x_n)$
…
$x_n \sqsubseteq F_n(x_1, \ldots, x_n)$

can be solved similarly by observing that $x \sqsubseteq y \Leftrightarrow x = x \sqcap y$ and rewriting the inequations.

Data-flow analysis

Overview

Data-flow analysis works on a control-flow graph and a lattice L. The lattice can either be fixed for all programs, or depend on the analysed one.

A variable v_n ranging over the values of L is attached to every node n of the CFG.

A set of (in)equations for these variables are then extracted from the CFG – according to the analysis being performed – and solved using the fixed point technique.

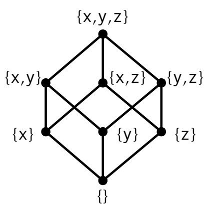
Example: liveness

As we have seen, liveness is a property that can be approximated using data-flow analysis.

The lattice to use in that case is $L = \{ \mathcal{P}(V), \subseteq \}$ where V is the set of variables appearing in the analysed program, and $\mathcal{P}$ is the power set operator (set of all subsets).

Example: liveness

For a program containing three variables x, y and z, the lattice for liveness is the following:



Example: liveness

To every node n in the CFG, we attach a variable v_n giving the set of variables live *before* that node.

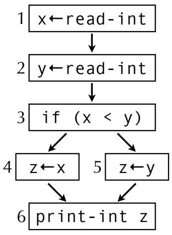
The value of that variable is given by:

$v_n = (v_{s1} \cup v_{s2} \cup \ldots \setminus \text{written}(n)) \cup \text{read}(n)$

where $s_1, s_2, \ldots$ are the successors of n, read(n) is the set of program variables read by n, and written(n) is the set of variables written by n.

Example: liveness

CFG



constraints

$v_1 = v_2 \setminus \{ x \}$
$v_2 = v_3 \setminus \{ y \}$
$v_3 = v_4 \cup v_5 \cup \{ x, y \}$
$v_4 = v_6 \cup \{ x \} \setminus \{ z \}$
$v_5 = v_6 \cup \{ y \} \setminus \{ z \}$
$v_6 = \{ z \}$

solution

$v_1 = \{ \}$
$v_2 = \{ x \}$
$v_3 = \{ x, y \}$
$v_4 = \{ x \}$
$v_5 = \{ y \}$
$v_6 = \{ z \}$

Fixed point algorithm

To solve the data-flow constraints, we construct the composite function F and compute its least fixed point by iteration.

$F(x_1, x_2, x_3, x_4, x_5, x_6) =$
$(x_2\setminus\{x\}, x_3\setminus\{y\}, x_4 \cup x_5 \cup \{x,y\}, x_6 \cup \{x\}\setminus\{z\}, x_6 \cup \{y\}\setminus\{z\}, \{z\})$

Iteration	x_1	x_2	x_3	x_4	x_5	x_6
0	{ }	{ }	{ }	{ }	{ }	{ }
1	{ }	{ }	{ x, y }	{ x }	{ y }	{ z }
2	{ }	{ x }	{ x, y }	{ x }	{ y }	{ z }
3	{ }	{ x }	{ x, y }	{ x }	{ y }	{ 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

$x_1 = x_2 = \ldots = x_n = \bot$
$q = [\ v_1, \ldots, v_n\ ]$
while ($q \neq []$)
  assume $q = [\ v_i, \ldots\ ]$
  $y = F_i(x_1, \ldots, x_n)$
  $q = q$.tail
  if ($y \neq x_i$)
    for ($v \in$ dep(v_i))
      if ($v \notin q$) q.append(v)
    $x_i = y$

Work-list example: liveness

q	x_1	x_2	x_3	x_4	x_5	x_6
$[v_1,v_2,v_3,v_4,v_5,v_6]$	{}	{}	{}	{}	{}	{}
$[v_2,v_3,v_4,v_5,v_6]$	{}	{}	{}	{}	{}	{}
$[v_3,v_4,v_5,v_6]$	{}	{}	{}	{}	{}	{}
$[v_4,v_5,v_6,v_2]$	{}	{}	{x,y}	{}	{}	{}
$[v_5,v_6,v_2,v_3]$	{}	{}	{x,y}	{x}	{}	{}
$[v_6,v_2,v_3,v_4]$	{}	{}	{x,y}	{x}	{y}	{}
$[v_2,v_3,v_4,v_5]$	{}	{}	{x,y}	{x}	{y}	{z}
$[v_3,v_4,v_5,v_1]$	{}	{x}	{x,y}	{x}	{y}	{z}
$[v_4,v_5,v_1]$	{}	{x}	{x,y}	{x}	{y}	{z}
$[v_5,v_1]$	{}	{x}	{x,y}	{x}	{y}	{z}
$[v_1]$	{}	{x}	{x,y}	{x}	{y}	{z}
[]	{}	{x}	{x,y}	{x}	{y}	{z}

Work-list improvements

In our liveness example, the work-list algorithm would have terminated in only six iterations if the initial queue had been reversed!

This is due to the fact that liveness analysis is a **backward** analysis: the value of variable v_n depends on the successors of n. For such analysis, it is better to organise the queue with the latest nodes first.

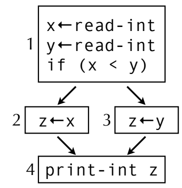
Working with basic blocks

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

When data-flow analysis is performed on a CFG composed of basic blocks, a variable is attached to every block, not to every instruction. Computing the result of the analysis for individual instructions is however trivial.

Liveness example revisited

CFG



constraints

$v_1 = v_2 \cup v_3 \setminus \{ x, y \}$
$v_2 = v_4 \cup \{ x \} \setminus \{ z \}$
$v_3 = v_4 \cup \{ y \} \setminus \{ z \}$
$v_4 = \{ z \}$

solution

$v_1 = \{ \}$
$v_2 = \{ x \}$
$v_3 = \{ y \}$
$v_4 = \{ z \}$

Analysis example #2: 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.

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
- 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 that use `x`, like `x+1`.

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.

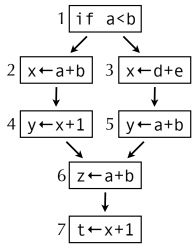
Then we derive constraints from the CFG nodes, which have the form:

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

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

Example

CFG



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

Analysis example #3: 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.

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*.

Equations

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

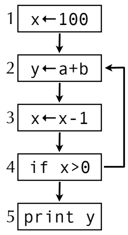
Then we derive constraints from the CFG nodes, which have the form:

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

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

Example

CFG



constraints	solution
$v_1 = v_2 \downarrow$`x`	$v_1 = \{$`a+b`$\}$
$v_2 = \{$`a+b`$\} \cup v_3 \downarrow$`y`	$v_2 = \{$`a+b`,`x-1`$\}$
$v_3 = \{$`x-1`$\} \cup v_4 \downarrow$`x`	$v_3 = \{$`x-1`$\}$
$v_4 = \{$`x>0`$\} \cup (v_5 \cap v_2)$	$v_4 = \{$`x>0`$\}$
$v_5 = \{\}$	$v_5 = \{\}$

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

Notation:
$S\downarrow$`x` $= S \setminus$ all expressions using variable `x`.

Analysis example #4: 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.

Intuitions

We will compute the set of reaching definitions *after* every node of the CFG. This 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 that define a variable defined by *n* itself, plus *n* itself.

Equations

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

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

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

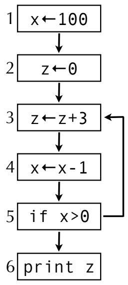
For a node *n* that is an assignment, the equation is more complicated:

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

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←`…

Example

CFG



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

v_n=set of reaching definitions after node n.

Notation:
$S\!\downarrow\!\mathtt{x} = S \setminus$ all nodes defining variable `x`.

Putting data-flow analyses to work

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 that 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.

CSE

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`.