

Foundations of Dataflow Analysis

15-745 Optimizing Compilers
Spring 2006

Peter Lee

Ingredients of a dataflow analysis

- * direction: forward or backward
- * flow/transfer function
- * combining (“meet”) operator
- * dataflow values

A thought experiment

- * As a thought experiment, consider programs with just one flow value
- * e.g., for reaching definitions, a program containing just one definition
- * e.g., for liveness analysis, a program with just one variable
- * e.g., for available expressions, a program with just one expression

Reaching definitions, again

combining operation (aka dataflow “meet” operator)

$$\text{in}(n) = \bigcup_{p \in \text{pred}(n)} \text{out}(p)$$

$$\text{out}(n) = \text{gen}(n) \cup (\text{in}(n) - \text{kill}(n))$$

flow function

$$\text{out}_n(i) = \text{gen}_n \vee (i - \text{kill}_n)$$

Dataflow values

- * For such simple programs, $in(n)$ and $out(n)$ would be elements of the set $\{0,1\}$, assuming a bit-vector representation
- * for RD: 1 = the def reaches
- * for LV: 1 = the var is live
- * for AE: 1 = the expr is available

Combining operation

- * When two or more control paths meet at a single node, the dataflow values from each path must be combined, according to the $in()$ equation (for forward analyses) or $out()$ equation (for backward analyses)
 - * RD: union / bit-wise \vee
 - * LV: union / bit-wise \vee
 - * AE: intersection / bit-wise \wedge
- } The "meet" operator

Recall: lattices

- * A **lattice** L is a (possibly infinite) set of values, along with \sqcup and \sqcap operations
- * $\forall x,y \in L, \exists$ unique w and z such that
 - * $x \sqcup y = w$ and $x \sqcap y = z$
- * $\forall x,y \in L, x \sqcup y = y \sqcup x$ and $x \sqcap y = y \sqcap x$
- * $\forall x,y,z \in L, (x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$ and $(x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$
- * $\exists \perp, \top \in L$, such that $\forall x \in L$,
 - * $x \sqcup \top = \top$ and $x \sqcap \perp = \perp$

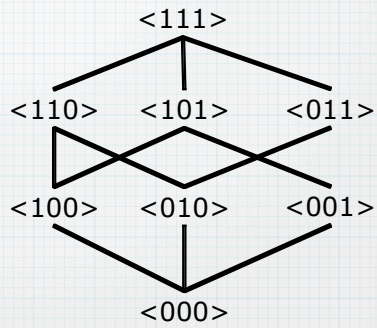
Two-point lattice

- * So, in fact, the values $\{0,1\}$, along with the bit-wise operations \vee and \wedge form a two-point lattice
- * \vee is the \sqcup ("join") operator
- * \wedge is the \sqcap ("meet") operator

1
|
0

Dataflow values are lattice elements

- * For our simple single-dataflow-value programs, the dataflow values are elements of the two-point lattice
- * And this is easily generalized to bit-vectors of arbitrary (but fixed) length



The lattice BV^3

Monotonic functions

- * The join and meet operations induce a **partial order** on the lattice elements
 - * $x \sqsubseteq y$ if and only if $x \sqcap y = x$
 - * reflexive, anti-symmetric, transitive
- * For a lattice L , a function $f: L \rightarrow L$ is **monotonic** if for all $x, y \in L$,
 - * $x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$
 - * i.e., larger args give larger results

RD is monotonic

$$\text{out}_n(i) = \text{gen}_n \vee (i - \text{kill}_n)$$

- * **Claim:** The flow function for reaching definitions is monotonic
- * **Proof** (for single-value programs) by contradiction:
 - * Suppose $\text{out}_n(1) = 0$, for some n
 - * Then $\text{gen}_n = 0$ and $\text{kill}_n = 1$
 - * But $\text{kill}_n = 1$ only if n is a definition, which would mean that $\text{gen}_n = 1$

In dataflow analysis, we require that all flow functions be monotone

Fixed points

- * Given lattice L and function $f:L \rightarrow L$, a **fixed point** of f is an element $z \in L$ such that $f(z)=z$
- * Furthermore, assuming there are no infinite ascending chains in L , iteration of f starting from \perp is guaranteed to find a fixed point
- * an **effective chain** wrt f is a sequence of lattice elements, $[f(i), f(f(i)), f(f(f(i))), \dots]$ for some $i \in L$

In dataflow analysis, we require that all flow functions be monotone and have only finite-length effective chains

Dataflow analysis framework

- * A control-flow graph with nodes $\{0,1,\dots,m\}$,
- * for each node, a corresponding monotonic flow function over lattice L , $\{f_0, f_1, \dots, f_m\}$,
- * a combining (aka "meet") operator, \otimes , and
- * an initial value, i_0 , giving the lattice value of the entry (or exit) block(s)

Iterative dataflow analysis

- * Then **forward iterative dataflow analysis** is the least solution in L to these equations:

$$\begin{aligned} \text{in}(\text{entry}) &= i_0 \\ \text{in}(n) &= \bigotimes_{p \in \text{pred}(n)} \text{out}(p), \text{ for non-entry nodes} \\ \text{out}(n) &= f_n(\text{in}(n)) \end{aligned}$$

For RD, LV, and AE, the f_n are monotonic and have finite effective chains, so the **iterative method will work**

Similarly for backward iterative dataflow analysis

Questions

- * Is it sound (i.e., are the results conservative)?
- * How good are the results?
- * How fast does it run?

How good is it?

- * Consider a path of execution that starts from the entry node and ends at node n :
 - * $\text{entry} \rightarrow b_1 \rightarrow b_2 \rightarrow \dots \rightarrow n$
- * The dataflow information for this path is given by composing the flow functions:
 - * $f_n(\dots(f_{b_2}(f_{b_1}(f_{\text{entry}}(i_0))))\dots)$

MOP

- * Then the “best” possible solution to a dataflow problem for node n is given by computing the dataflow information for all possible paths from entry to n , and then combining them with \otimes
- * in general there will be an infinite number of possible paths to n
- * This is called the **meet-over-all-paths solution**

IDA is conservative

- * Thm [Kildall73]: The MOP solution for block n is \sqsubseteq the solution given to $\text{in}(n)$ by iterative dataflow analysis
 - * a related theorem by [Cousot&Cousot76]
- * In other words, the iterative dataflow analysis solution is a conservative approximation to the “best” solution

Distributive lattices

- * BV^n is a **distributive lattice**
- * L is distributive if $\forall x, y, z \in L$,
 - * $(x \sqcup y) \sqcap z = (x \sqcap z) \sqcup (y \sqcap z)$, and
 - * $(x \sqcap y) \sqcup z = (x \sqcup z) \sqcap (y \sqcup z)$

BV^n analyses are precise

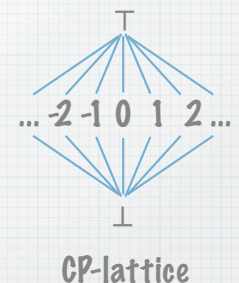
- * Thm [Kildall73]: For iterative dataflow analysis over a distributive lattice, $\text{in}(n)$ is the MOP solution for n
- * Thus, reaching definitions, liveness, and available expressions are all solved precisely by iterative dataflow analysis

Not all DAs are distributive

- * There are important dataflow analyses that use lattices that are not distributive
- * Consider, for example, constant-propagation analysis...

Constant-propagation analysis

- * For a single-variable program:
 - * direction: forward
 - * dataflow value: element of the CP-lattice
 - * meet operator: CP-lattice \sqcap
 - * initial entry value: \perp



\perp means "uninitialized variable"

\top means "not a constant"

Constant-propagation analysis

- * For multi-variables we have the lattice CP^* :
- * elements: functions $f: Var \rightarrow CP\text{-lattice}$
- * $\perp_{CP^*}: \perp_{CP^*}(v) = \perp$, all v
- * $\top_{CP^*}: \top_{CP^*}(v) = \top$, all v
- * $(f \sqcup_{CP^*} g)(v) = f(v) \sqcup g(v)$, all v

CP is not distributive

- * The CP -lattice and CP^* are not distributive
- * $(1 \sqcup 2) \sqcap 3 = \top \sqcap 3 = 3 \neq$
- * $(1 \sqcap 3) \sqcup (2 \sqcap 3) = \perp \sqcup \perp = \perp$
- * In practical terms, this means that the order in which the nodes are visited can affect the precision of the results

Not always finite chains

- * Sometimes we will want to use lattices that have infinite ascending chains
- * In order to use such lattices in an iterative dataflow analysis, we will have to ensure that the flow functions still have only finite effective chains
- * Standard example: range analysis

Range analysis

- * With range analysis, we try to determine the possible range of integer values of a variable
- * elts: \perp , \top , and $[m, n]$ where $m \leq n$
- * $[m, n] \sqcup [m', n'] = [\min(m, m'), \max(n, n')]$
- * A forward analysis with \sqcup as the meet operator
- * How to deal with the infinite chains?

MOP is not always “best”

- * For RD, LV, and AE, the flow functions are simple in the sense of being constant-time (or actually $O(N)$, for N -variable programs)
- * But the flow functions can, in principle, be arbitrarily complicated, even involving theorem proving
 - * e.g.: $v = x \bmod y$, for prime x ...

How fast?

- * Consider reaching definitions
 - * N nodes, so N possible definitions
 - * N^2 bits
 - * in worst case, each iteration changes one bit
 - * Thus, $O(N^3)$, assuming constant-time flow functions (per bit)
 - * Ex: Construct a worst-case example

Why do basic blocks help?

- * In terms of worst-case complexity, basic blocks do not make a difference
- * However, we usually assure proper directionality within blocks, effectively assuring that the dataflow works in linear time

Reduction Methods

Iterative dataflow analysis is easy

- * Iterative dataflow analysis is relatively easy to implement
- * Hence, its popularity in optimizing compilers
- * But it can be slow to execute
- * With the rise in popularity of Just-In-Time compilers, dataflow analysis methods based on reducibility of flow graphs have re-gained interest

Reduction methods

- * In reduction methods, we calculate, symbolically, a flow function for the entire procedure body
 - * usually this can be done in linear time
- * This is called a “reduction” because, in essence, the flow graph is reduced to a single node
- * Then, in practice, a fixed point can be found in a small number (often just one) iterations

Next time...

- * Overview of reduction methods
- * Start on loop optimizations