Lecture Notes on Monotone Frameworks & Abstract Interpretation

15-411: Compiler Design
André Platzer

Lecture 26
November 30, 2010

1 Introduction

More information on dataflow analysis and monotone frameworks can be found in [NNH99]. More information on abstract interpretation can be found in [CC77, CC79] and [WM95, Chapter 10].

2 Forward May Dataflow Analysis

A forward may dataflow analysis follows the principle shown in Fig. 1.

There, $A_\circ(l)$ is the information that holds at the entry of a block and we compute it as a union of all that may hold at the previous blocks. Thus $A_\circ(l)$ will hold anything that may hold at any previous block. $A_\bullet(l)$ is the information that holds at the exit of a block. $kill(l)$ holds the information that we remove from the input. $gen(l)$ holds the information that we add to the input. We compute $A_\bullet(l)$ as a function of the information $A_\circ(l)$ holding at the entry, minus those that we remove ($kill(l)$) plus those that we add ($gen(l)$).

If we choose $\iota = \emptyset$ for a may analysis, we obtain the least fixed point.

Recall, for example, the reaching definitions analysis, which is a forward may analysis with the choice in Table 1. It analyses which assignments may have been made before but have not been overwritten yet.

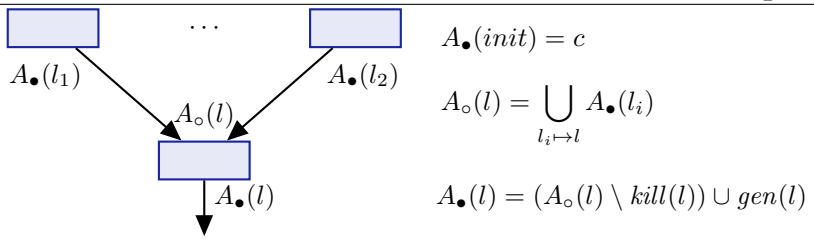


Figure 1: Dataflow analysis schema for forward may analysis

Table 1: Forward may analysis definitions for reaching definitions

Statement l	$gen(l)$	$kill(l)$
$init$	$A_\bullet(init) = Lbl$	
$l : x \leftarrow a \odot b$	$\{l\}$	$\{l : def(l, x)\}$
$l : x \leftarrow *a$	$\{l\}$	$\{l : def(l, x)\}$
$l : *a \leftarrow b$	$\emptyset$	$\emptyset$
goto l'	$\emptyset$	$\emptyset$
if $a > b$ **goto** l'	$\emptyset$	$\emptyset$
$l' :$	$\emptyset$	$\emptyset$
$l : x \leftarrow f(p_1, \ldots, p_n)$	$\{l\}$	$\{l : def(l, x)\}$

3 Forward Must Dataflow Analysis

Forward may dataflow analysis looks for information that may hold on some of the paths. If, instead, we need information about something that must hold on all of the paths, we use a forward must dataflow analysis instead (Figure 2). If we choose ι as everything for a must analysis, we obtain the greatest fixed point.

Recall, for instance, the dataflow equations for the available expression analysis from the optimization lecture, which we show again in Table 2. Other forward must analysis includes dominator analysis to determine which statements dominate the program point, i.e., must have been executed before.

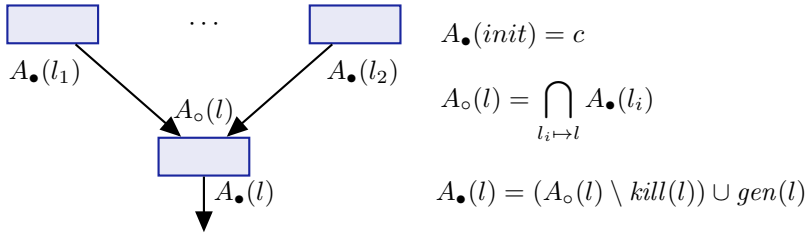


Figure 2: Dataflow analysis schema for forward must analysis

Table 2: Forward must analysis definitions for available expressions (recall)

Statement l	$gen(l)$	$kill(l)$
$init$	$A_\bullet(init) = \emptyset$	
$x \leftarrow a \odot b$	$\{a \odot b, a, b\} \setminus kill(l)$	$\{e \ : \ e$ contains $x\}$
$x \leftarrow *a$	$\{*a\} \setminus kill(l)$	$\{e \ : \ e$ contains $x\}$
$*a \leftarrow b$	$\emptyset$	$\{*z \ : \$ for all $z\}$
goto l'	$\emptyset$	$\emptyset$
if $a > b$ **goto** l'	$\emptyset$	$\emptyset$
l' :	$\emptyset$	$\emptyset$
$x \leftarrow f(p_1, \ldots, p_n)$	$\emptyset$	$\{e \ : \ e$ contains x or is $* z\}$

4 Backward May Dataflow Analysis

Following the control flow forward is not the only direction that makes sense. Backward dataflow analysis follows the control flow backwards instead. It again comes in two flavors: backward may and backward must dataflow analysis. For backward dataflow analysis, we no longer initialize the analysis at the initial node, but at all final nodes instead, because we follow the control flow backwards from the final nodes to the beginning.

Live variable analysis is an example of a backward may analysis (Figure 3), i.e., which variables may be live, i.e., there is a path to a use without redefinition. See Table 3.

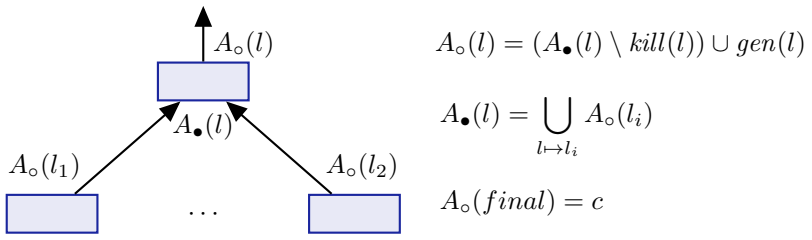


Figure 3: Dataflow analysis schema for backward may analysis

Table 3: Backward may analysis definitions for live variables

Statement l	$gen(l)$	$kill(l)$
$final$	$A_\bullet(final) = \emptyset$	
$x \leftarrow a \odot b$	$\{a, b\}$	$\{x\}$
$x \leftarrow *a$	$\{a\}$	$\{x\}$
$*a \leftarrow b$	$\{a, b\}$	$\emptyset$
goto l'	$\emptyset$	$\emptyset$
if $a > b$ **goto** l'	$\{a, b\}$	$\emptyset$
l' :	$\emptyset$	$\emptyset$
$x \leftarrow f(p_1, \ldots, p_n)$	$Vars(\{p_1, \ldots, p_n\})$	$\{x\}$

5 Backward Must Dataflow Analysis

Very busy expressions is an example of a backward must analysis (Figure 4), i.e., which expressions will be used on every path before any of its variables is redefined. Very busy expressions are needed for partial redundancy elimination (PRE) and can be useful for determining which variable to keep in a register instead of spilling. Variables that are used again on every path may be more useful to keep in a register than those that are only used on one path. See Table 4.

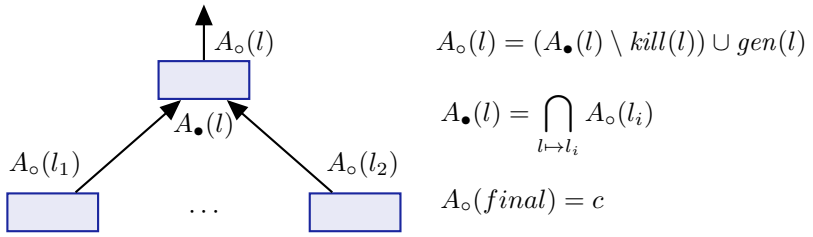


Figure 4: Dataflow analysis schema for backward must analysis

Table 4: Backward must analysis definitions for very busy expressions

Statement l	$gen(l)$	$kill(l)$
$final$	$A_\bullet(final) = \emptyset$	
$x \leftarrow a \odot b$	$\{a \odot b\}$	$\{e \ : \ e$ contains $x\}$
$x \leftarrow *a$	$\{*a\}$	$\{e \ : \ e$ contains $x\}$
$*a \leftarrow b$	$\{b\}$	$\{e \ : \ e$ contains $*z$ for any $z\}$
goto l'	$\emptyset$	$\emptyset$
if $a > b$ **goto** l'	$\{a, b\}$	$\emptyset$
l' :	$\emptyset$	$\emptyset$
$x \leftarrow f(p_1, \ldots, p_n)$	$\emptyset$	$\{e \ : \ e$ contains x or any $*z\}$

6 Monotone Frameworks

Even though all of them are different, the forward/backward may/must dataflow analysis are nevertheless very similar. They all follow a general pattern:

$$A_\circ(\ell) = \begin{cases} \iota & \text{if } \ell \in E \\ \bigsqcup\{A_\bullet(\ell') \ : \ (\ell', \ell) \in F\} & \text{otherwise} \end{cases}$$
$$A_\bullet(\ell) = f_\ell(A_\circ(\ell))$$

where, depending on the specific analysis:

- the operator $\bigsqcup$ is either $\bigcup$ for information from any source or $\bigcap$ for information joint to all sources
- the flow relation F is either the forward control flow or the backward control flow
- the initialization set E is either the initial block or the set of final nodes
- ι specifies the starting point of the analysis at the initial or final nodes
- f_ℓ is the transfer function for the node, which, in the previous examples is always of the special form

$$f_\ell(X) = (X \setminus kill(\ell)) \cup gen(\ell)$$

More formally, the property that we are analyzing is part of a *property space* L. This space L could be the set of all sets of variables $\wp(Vars)$, if we are looking for the set of all live variables. Or, for available expressions, it could be the set of all sets of expressions $\wp(Expr)$ ordered by $\supseteq$. Or, in fairly advanced analyses, we might even be tempted to try the set of all mappings $Vars \to \mathbb{Z}^2$ from variables to intervals, if we are trying to find interval bounds for each variable. The latter scenario is more difficult, though.

For the property space and the way how property values flow through the control flow, we need a number of assumptions.

Definition 1 *A* monotone framework *consists of*

1. *a Noetherian complete semi-lattice L: a set L with a partial order $\sqsubseteq$ that is complete, i.e., such that* each *subset $Y \subseteq L$ has a least upper bound $\bigsqcup Y$. A lattice is Noetherian iff it satisfies the condition that each ascending chain*

$$a_1 \sqsubseteq a_2 \sqsubseteq a_3 \sqsubseteq ...$$

   is finite, i.e., there is an n such that $a_n = a_{n+1} = a_{n+2} = ...$

2. *a set $\mathcal{F}$ of monotone functions $f : L \to L$ that contains the identify function $id : L \to L; a \mapsto a$ and is closed under composition, i.e., if $f, g \in \mathcal{F}$ then the composition $f \circ g \in \mathcal{F}$. A function $f : L \to L$ is monotone iff*

$$a \sqsubseteq b \text{ implies } f(a) \sqsubseteq f(b)$$

A "distributive" framework *is a monotone framework where each $f \in \mathcal{F}$ is distributive (or, more precisely, a homomorphism)*

$$f(a \sqcup b) = f(a) \sqcup f(b)$$

Note that we use the binary operator notation $a \sqcup b$ as an abbreviation for the more verbose $\bigsqcup\{a, b\}$. In addition, we denote the least upper bound $\bigsqcup \emptyset$ by $\bot$, which is the least element of the semi-lattice L.

Definition 2 *The analysis equations corresponding to a monotone framework are*

$$A_\circ(\ell) = \bigsqcup\{A_\bullet(\ell') \ : \ (\ell', \ell) \in F\} \ \sqcup \ \begin{cases} \iota & \text{if } \ell \in E \\ \bot & \text{if } \ell \notin E \end{cases} \tag{1}$$

$$A_\bullet(\ell) = f_\ell(A_\circ(\ell)) \tag{2}$$

7 Solving Monotone Framework Equations as Least Fixed Points

A simple way of solving the analysis equations of a monotone framework works by iteratively updating the left hand side of (1) to match the right hand side of (1) until nothing changes anymore. This is the worklist algorithm[1]:

[1] In this context this algorithm is often called "Maximum" Fixed Point, which is rather confusing for a least fixed point.

Table 5: Dataflow analysis examples as monotone frameworks

	L	$\sqsubseteq$	$\bigsqcup$	$\bot$	ι	E	F
For-may: Reach def	$\wp(Lbl)$	$\subseteq$	$\bigcup$	$\emptyset$	Lbl	$init$	$flow$
For-must: Available expr	$\wp(Exp)$	$\supseteq$	$\bigcap$	Exp	$\emptyset$	$init$	$flow$
Back-may: Live variables	$\wp(Var)$	$\subseteq$	$\bigcup$	$\emptyset$	$\emptyset$	$final$	$flow^{-1}$
Back-must: Very busy expr	$\wp(Exp)$	$\supseteq$	$\bigcap$	Exp	$\emptyset$	$final$	$flow^{-1}$

$$f_\ell(l) = (l \setminus kill(\ell)) \cup gen(\ell)$$
$$\mathcal{F} = \{f : L \to L \ : \ f(l) = (l \setminus kill) \cup gen \text{ for some } kill, gen\}$$

```
W := F      // working list
for ℓ ∈ F ∪ E
    if (ℓ ∈ E) then A∘[ℓ] := ι else A∘[ℓ] := ⊥
for (ℓ', ℓ) ∈ W
    W := W \{(ℓ', ℓ)}
    if f_ℓ'(A∘[ℓ']) ⋢ A∘[ℓ] then
        A∘[ℓ] := A∘[ℓ] ⊔ f_ℓ'(A∘[ℓ'])
        W := {(ℓ, ℓ'') : (ℓ, ℓ'') ∈ F} ∪ W
end for
```

This algorithm computes the least fixed point of (1), because it starts at the bottom $\bot$ of the semi-lattice and successively follows the fixed point condition. It also always terminates. Let's convince ourselves why.

Theorem 3 (Kleene fixed point theorem) *If L is a complete partial order and $f : L \to L$ is a Scott-continuous function (i.e., $\bigsqcup f(Y) = f(\bigsqcup Y)$ for every subset $Y \subseteq L$ with a supremum $\bigsqcup Y \in L$), then the least fixed point μf of f is*

$$\mu f = \bigsqcup \{f^n(\bot) \ : \ n \in \mathbb{N}\}$$

The Kleene fixed point theorem is very useful, because it shows that successive iteration like we are doing in the worklist algorithm really yields the least fixed point. Yet are we in a position to use the theorem? It doesn't quite look like it. We have montone transfer functions but need Scott-continuous functions. Every Scott-continuous function $f : L \to L$ is monotone:

$$a \sqsubseteq b \ \Rightarrow \ \bigsqcup\{f(a), f(b)\} = f(\bigsqcup\{a, b\}) = f(b) \ \Rightarrow \ f(a) \sqsubseteq f(b)$$

But in monotone frameworks, we deal with monotone functions $f : L \to L$ and need to know whether they are Scott-continuous. Now fortunately, the semi-lattice L underlying monotone frameworks is actually Noetherian. Thus, if we start with any set $Y \subseteq L$ and want to relate $\bigsqcup f(Y)$ and $f(\bigsqcup Y)$, we first note that Y cannot contain an infinite ascending chain but can only contain a finite ascending chain:

$$a_1 \sqsubseteq a_2 \sqsubseteq a_3 \sqsubseteq ... \sqsubseteq a_n$$

Now by monotonicity of f we know

$$f(a_1) \sqsubseteq f(a_2) \sqsubseteq f(a_3) \sqsubseteq ... \sqsubseteq f(a_n)$$

Thus,

$$\bigsqcup\{f(a_1), \ldots, f(a_n)\} = f(a_n) = f(\bigsqcup\{a_1, \ldots, a_n\})$$

The same argument holds for all ascending chains in Y. The ends of all those finite ascending chains have least upper bounds, because L is a lattice. Because L is Noetherian, even those extensions can only give finite ascending chains.

The only trouble is that the function f we are using in the algorithm to form a fixed point is not just one of the transfer functions f_ℓ, which we know to be monotone. The overall function f is really

$$f(Z)(\ell) := \bigsqcup\{f_{\ell'}(Z(\ell')) \ : \ (\ell', \ell) \in F\} \sqcup \begin{cases} \iota & \text{if } \ell \in E \\ \bot & \text{if } \ell \notin E \end{cases}$$

$$f(Z) := \left\{\left(\ell, \bigsqcup\{f_{\ell'}(Z(\ell')) \ : \ (\ell', \ell) \in F\}\right) \ : \ \ell \in E \cup F\right\} \\ \sqcup \{(\ell, \iota) \ : \ \ell \in E\} \sqcup \{(\ell, \bot) \ : \ \ell \in F \setminus E\}$$

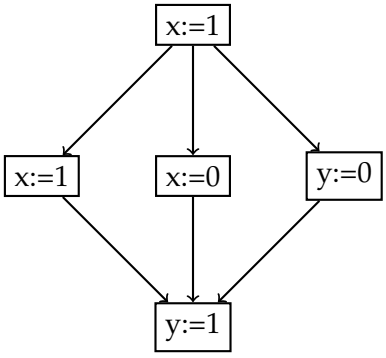
Now this function is more complicated because it has the nodes ℓ and ℓ' as extra arguments, so the domain is a slightly different one and things become more complicated. Nevertheless, the principles above still apply and we can see that f is monotone, because the result only increases at every point if the input increases.

Why does the worklist algorithm terminate? That's easy to see if the semi-lattice L is finite, because there can only be finitely many monotone changes to the sets then. What if the property space L is infinite? Well the algorithm still terminates, because $A_\circ[\ell]$ increases at its assignment, and it can only increase to a finite ascending chain, not an infinite one, because L is Noetherian.

8 Abstract Interpretation

Abstract interpretation generalizes the theory of monotone frameworks and dataflow analysis to a general principle of analyzing programs by defining an abstract semantics for it [CC77, CC79, WM95]. In order to show the principle of abstract interpretation, without having to dig too much into the details, we consider an example where we abstractly interpret a program but still keep using monotone frameworks.

Suppose we want to check the property whether a variable x may be 0, which is a principle that can be useful for null pointer exception tests. As domain L for this we just choose the Boolean lattice $\{true, , false\}$. The operator $\bigsqcup$ is just logical disjunction ($\vee$). The flow relation is the forward control flow. Initialization is $false$, say. Transfer functions at the nodes make sense to choose from the constant functions $true$, $false$ and the identity function id.



By fixed-point iteration on the above example we find that $x = 1$ is possible after the program terminates. For a must analysis, instead, we would get that $x = 1$ is not necessary.

For multiple variables, we can choose a cartesian product $\{true, false\}^n$ of the Boolean lattice and use projections to coordinates as further transfer functions for copying the value for y over to x at a move $x := y$.

Another example is an abstract interpretation that performs general analysis for constant propagation. The property space has the form $\{x = \bot, x =?\} \cup \{x = v \ : \ v \in \mathbb{Z}\}$, where $\bot$ means is the bottom of the semilattice for undefined, $x =?$ means that x has nondeterministic values and $x = v$ for a number v means that we can be certain that x will always have value v at this program point. Let's look at an example. We initialize with no information ($\bot$) at all points, except the program init block, where we start with a nondeterministic initial value $i =?$:

```
{i=?,j=?,k=?}
i = 5; j = 0; k = 0;
{i=⊥,j=⊥,k=⊥}
while (j <= i) {
  {i=⊥,j=⊥,k=⊥}
  i = i + 2; k = k + j; j = j + 1
  {i=⊥,j=⊥,k=⊥}
  i = i − 2
  {i=⊥,j=⊥,k=⊥}
}
{i=⊥,j=⊥,k=⊥}
```

Now we can execute the first line in the abstract semantics and then enter the loop in the abstract semantics and execute the loop body once

```
{i=?,j=?,k=?}
i = 5; j = 0; k = 0;
{i=5,j=0,k=0}
while (j <= i) {
  {i=5,j=0,k=0}
  i = i + 2; k = k + j; j = j + 1
  {i=7,j=1,k=0}
  i = i − 2
  {i=5,j=1,k=0}
}
{i=⊥,j=⊥,k=⊥}
```

With those abstract values, we will repeat the loop, but we have to merge the previous information {i=5,j=0,k=0} with the current information {i=5,j=1,k=0} and find a joint representation in the property space lattice by the $\bigsqcup$ operator, giving {i=5,j=?,k=0}. Then we execute the loop body

```
{i=?,j=?,k=?}
i = 5; j = 0; k = 0;
{i=5,j=0,k=0}
while (j <= i) {
  {i=5,j=?,k=0}
  i = i + 2; k = k + j; j = j + 1
  {i=7,j=?,k=?}
  i = i − 2
  {i=5,j=?,k=?}
}
```

```
{i=⊥,j=⊥,k=⊥}
```

Again, merging the property values by the $\sqcup$ operator and executing the loop body gives

```
{i=?,j=?,k=?}
i = 5; j = 0; k = 0;
{i=5,j=0,k=0}
while (j <= i) {
  {i=5,j=?,k=?}
  i = i + 2; k = k + j; j = j + 1
  {i=7,j=?,k=?}
  i = i − 2
  {i=5,j=?,k=?}
}
{i=5,j=?,k=?}
```

Here the property value at the loop entry didn't change, so we can propagate to the loop exit and the analysis terminates. Now we know, as good as our abstract semantics could represent, what values the variables can have at the various program points.

References

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282, 1979.

[NNH99] F. Nielson, H. R. Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer, 1999.

[WM95] Reinhard Wilhelm and Dieter Maurer. *Compiler Design*. Addison-Wesley, 1995.