Lecture Notes on
Abstract Dataflow Analysis

15-411: Compiler Design
André Platzer

Lecture 26
November 30, 2010

1 Introduction

We have seen numerous dataflow analyses in class. They have a lot more in common than one might think.

More information on dataflow analysis and monotone frameworks can be found in [NNH99].

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.
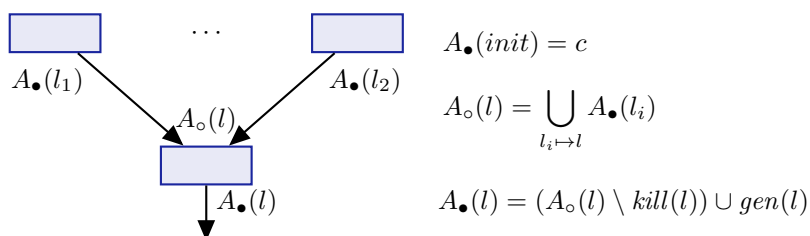
$$A_\bullet(init) = c$$

$$A_\circ(l) = \bigcup_{l_i \mapsto l} A_\bullet(l_i)$$

$$A_\bullet(l) = (A_\circ(l) \setminus kill(l)) \cup gen(l)$$

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.
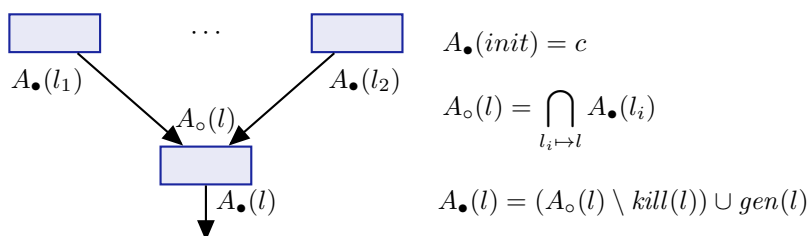
$$A_\bullet(init) = c$$

$$A_\circ(l) = \bigcap_{l_i \mapsto l} A_\bullet(l_i)$$

$$A_\bullet(l) = (A_\circ(l) \setminus kill(l)) \cup gen(l)$$

Figure 2: Dataflow analysis schema for forward must analysis

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

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

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.
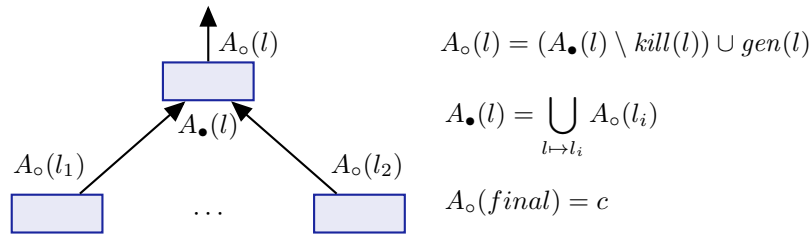
$$A_\circ(l) = (A_\bullet(l) \setminus kill(l)) \cup gen(l)$$

$$A_\bullet(l) = \bigcup_{l \mapsto l_i} A_\circ(l_i)$$

$$A_\circ(final) = c$$

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.

6   What These Dataflow Analyses Have in Common

Even though all of them are different, the forward/backward may/must dataflow analysis are nevertheless very similar. They all follow a general
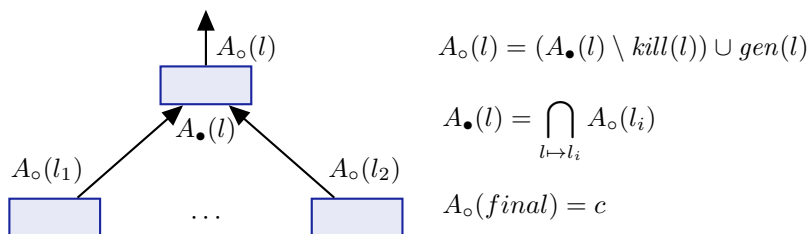
Figure 4: Dataflow analysis schema for backward must analysis

$$A_\circ(l) = (A_\bullet(l) \setminus kill(l)) \cup gen(l)$$

$$A_\bullet(l) = \bigcap_{l \mapsto l_i} A_\circ(l_i)$$

$$A_\circ(final) = c$$

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 $*z\}$

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. We will investigate those assumptions and the general principle behind dataflow analysis in the next lecture.

References

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