3.5 Monotone data flow systems

3.5.1 An informal introduction

Virtually all data flow problems can be modeled and solved in a uniform way by using the concept of monotone data flow systems. We shall illustrate the basic ideas for the reaching definitions problem RD (see Definition 3.19).

1) Definition of the data flow information set L
L is the set of all data flow information items that are relevant for the solution of the given problem.

For every \( n \in N \), \( RD(n) \) is a subset of \( S.DEFS \). Thus the powerset \( \mathcal{P}(S.DEFS) \) is the data flow information set \( L \) in this case.

2) The effect of joining paths
Assume that \( n \in N \) is given, \( \text{pred}(n) = \{n_1, n_2\} \) and data flow information \( X_i \) leaves \( n_i \) (\( i = 1,2 \)). How can we compute the information to be associated with \( n \) from \( X_1 \) and \( X_2 \)?

For RD, the question can be answered as follows: any definition that is in \( X_1 \) or \( X_2 \) reaches \( n \); so we must form the union of \( X_1 \) and \( X_2 \).

3) The effect of basic blocks
Each basic block \( n \) must be associated with a function \( f_n \), which specifies the effect an execution of \( n \) has on the data flow information.

Assume that \( X \subseteq S.DEFS \) is a set of definitions that reaches \( n \). How do we determine the set \( X' \subseteq S.DEFS \) of definitions that leaves \( n \)? First, a definition \( S \) in \( X \) will be passed through \( n \) iff \( S \) is preserved in \( n \). Secondly, all definitions of \( n \) which are outward exposed reach the end of \( n \). Formally, we obtain:

\[
X' = f_n(X) = (X \cap S.PRE(n)) \cup S.DEF(n)
\]

for every \( X \subseteq S.DEFS \)

4) Solution of the problem by an iterative algorithm
Now, after a proper initialization of nodes with data flow information, the problem can be solved by propagating data flow information iteratively through the flowgraph. This process terminates when the propagation does not yield new information for any node of the graph. The iterative algorithm for the reaching definitions problem is given in Algorithm 3.4.
ALGORITHM 3.4: Reaching definitions

**Input**
1. Program $P$.
2. Data flow information set $L = \mathcal{P}(S_{\text{DEFS}})$.
3. Program graph $G = (N,E,s)$.
4. For each $n \in N$: $S_{\text{PRE}}(n)$ and $S_{\text{DEF}}(n)$.

**Output** For each $n \in N$: $RD(n)$.

**Method**

```latex
\begin{verbatim}
/* The algorithm uses the following variables:
new: a temporary variable, to which elements of $L$ are assigned
stable: a boolean variable, which is used to test for stabilization of
the algorithm
INF(n) for every $n \in N$: a variable that stores the actual data flow information
associated with node $n$. The final value of $INF(n)$ specifies the result of
the algorithm for that node. */

begin
/* Initialization */
for every $n \in N$ do $INF(n) := \emptyset$ end for;
/* Iteration */
repeat
stable := true;
for every $n \in N$ do
new := \bigcup_{n' \in \text{pred}(n)} ((INF(n') \cap S_{\text{PRE}}(n')) \cup S_{\text{DEF}}(n'));
if $new \neq INF(n)$ then $INF(n) := new; stable := false$
fi
end for
until stable;
for every $n \in N$ do $RD(n) := INF(n)$ end for
end
```

In general, we require the data flow information set $L$ to be a bounded
semilattice, and the functions $f_n$ to be monotone functions on $L$. The effect
of joining paths is modeled by the meet operation of the semilattice. The
optimum solution of the data flow problem is defined for each $n \in N, n \neq s$,
as the value obtained by (a) determining the effect of every path $\pi$ from $s$ to $n$
by propagating data flow information along $\pi$, and (b) applying the meet
operation of $L$ to the set of values determined in (a). The iterative
algorithm can be shown to terminate for all instances of the problem, and to yield
either the optimum solution or a conservative approximation thereof.

We now define the concepts needed.
3.5.2 Semilattices

**Definition 3.20** A semilattice is a set $L$ with a binary meet operation $\wedge$ such that for all $a,b,c \in L$:

1. $a \wedge a = a$ (idempotent)
2. $a \wedge b = b \wedge a$ (commutative)
3. $a \wedge (b \wedge c) = (a \wedge b) \wedge c$ (associative)

If we want to make the association of $L$ with $\wedge$ explicit, we write $(L, \wedge)$ for the semilattice.

A semilattice has a **zero element** $0$ iff $a \wedge 0 = 0$ for every $a \in L$. $L$ has a **one element** $1$ iff $a \wedge 1 = a$ for every $a \in L$. If $0$ or $1$ exists, then it is unique.

**Definition 3.21** Let $(L, \wedge)$ denote a semilattice, and let $a,b$ be arbitrary elements of $L$. We define a relation $\leq$ in $L$:

$$a \leq b : \iff a \wedge b = a$$

Whenever we use the symbol `\leq` in the context of a semilattice, we will mean the relation defined here. Based on this definition, the relations $<, \geq, >$ can be obtained in the usual way.

The following lemma can be easily verified.

**Lemma 3.4** Let $(L, \wedge)$ denote a semilattice and $\leq$ the relation introduced in Definition 3.21. Then $\leq$ is a partial order on $L$.

Let $a_1, a_2, \ldots$ denote a sequence of elements from a semilattice $L$. This sequence is called a **chain** iff $a_i > a_{i+1}$ for all $i = 1, 2, \ldots$.

**Definition 3.22** A semilattice $L$ is **bounded** iff for every $a$ in $L$ there exists an $a_c \in \mathbb{N}$ such that the length of every chain beginning with $a$ is at most $c_a$.

---

**Example 3.11**

Let $M$ denote an arbitrary finite set.

1. $(\mathcal{P}(M), \cap)$ is a bounded semilattice with both $0$ and $1$ ($\emptyset$ and $M$, respectively). The relation $\leq$ is the set-theoretic inclusion $\subseteq$. 

---
Scalar Analysis

(2) \((θ(M), \cup)\) is a bounded semilattice with 0 and 1 elements, where 0 is \(M\) and 1 is \(∅\). The relation \(≤\) corresponds to the inverse inclusion, \(≤^{-1}\).

The meet operation can be readily extended to an arbitrary number \(m \geq 1\) of elements by defining:

\[
\bigwedge_{1 \leq i \leq m} a_i := a_1 \wedge a_2 \wedge \ldots \wedge a_m
\]

If \(L\) is bounded, then \(\wedge\) can be further extended to countably infinite sets: Let \(M \subseteq L\) denote such a set. Then:

\[
\bigwedge_{a \in M} a := \lim_{m \to \infty} \bigwedge_{1 \leq i \leq m} a_i
\]

The limit exists and is equal to

\[
\bigwedge_{1 \leq i \leq q} a_i
\]

for some \(q \in \mathbb{N}\).

In all of the following we shall assume that \((L, \wedge)\) is a bounded semilattice with 0 and 1.

3.5.3 Monotonic functions and their largest fixpoint

We have mentioned before that the effect a basic block has on data flow information will be modeled by a function \(f: L \to L\). These functions must be monotonic in the sense defined below. A special subclass of monotonic functions for which the iterative algorithm produces precise results is the class of distributive functions.

**Definition 3.23**

(1) A total function \(f: L \to L\) is monotonic :\(\Rightarrow\) for all \(a, b \in L\):

\[f(a \wedge b) \leq f(a) \wedge f(b)\]

(2) A total function \(f: L \to L\) is distributive :\(\Rightarrow\) for all \(a, b \in L\):

\[f(a \wedge b) = f(a) \wedge f(b)\]

It can be easily seen that a function is monotonic iff for all \(a, b \in L\) \(a \leq b\) implies \(f(a) \leq f(b)\).

A fixpoint of a monotonic function \(f: L \to L\) is a value \(a \in L\) such that \(f(a) = a\). The following theorem provides the foundation for iterative data flow analysis algorithms.
Theorem 3.2 Let \( L \) denote a semilattice and \( f : L \rightarrow L \) a monotonic function. Then there exists a \( t \geq 0 \) such that \( f^{t+1}(1) = f^t(1) \). \( f^t(1) \) is the greatest fixpoint of \( f \).

Proof Since \( f \) is monotonic, the sequence \( 1, f(1), f(f(1)), \ldots \) is a monotone descending sequence of elements from \( L \). \( L \) is bounded, so there is a \( t \) such that \( f(f^t(1)) = f^t(1) \). Clearly, \( f^t(1) \) is a fixpoint of \( f \). Now let \( a \) denote an arbitrary fixpoint of \( f \). From \( a \leq 1 \) and the monotonicity property we obtain \( f^r(a) \leq f^r(1) \) for all \( r \geq 0 \); as \( f^r(a) = a \) for all \( r \), we immediately obtain \( a = f^t(1) = fp \). Therefore, \( fp \) is the greatest fixpoint of \( f \).

The fixpoint algorithm to determine the greatest fixpoint of a monotonic function is derived immediately from Theorem 3.2.

ALGORITHM 3.5: Greatest fixpoint of a monotonic function

Input  (1) Semilattice \((L, \land)\).
        (2) \( f : L \rightarrow L \), monotonic.

Output The greatest fixpoint, \( fp \), of \( f \).

Method

begin
  \( a := 1 \);
  while \( f(a) < a \) do \( a := f(a) \) end while;
  \( fp := a \)
end

3.5.4 Monotone data flow systems and their optimal solution

Definition 3.24

(1) A monotone function space for a semilattice \( L \) is a set \( F \) of monotonic functions which (i) contains the identity function \( id \), (ii) is closed under function composition, and (iii) satisfies the following condition: for each \( a \in L \) there is an \( f \in F \) such that \( f(0) = a \).

(2) A distributive function space is a monotone function space in which all functions are distributive.

The conditions imposed on monotone function spaces are motivated (i) by the need to model basic blocks which do not change data flow information,
Scalar Analysis

(ii) by the necessity of modeling the effect of paths in the flowgraph, and
(iii) by technical considerations.

Definition 3.25 A monotone data flow system (MDS) is a tuple \( \Omega = (L, \wedge, F, G, FM) \), where:

1. \((L, \wedge)\) is a bounded semilattice with 0 and 1.
2. \(F\) is a monotone function space for \(L\).
3. \(G = (N, E, s)\) is a program graph.
4. \(FM: N \to F\) is a total function.

An MDS is distributive iff \(F\) is a distributive function space for \(L\).

The function \(FM\) associates with every node of the flowgraph a function in \(F\). For every \(n \in N\) we will use the shorthand \(f_n\) instead of \(FM(n)\). Let \(\pi = (n_1, n_2, \ldots, n_k, n_{k+1})\), \(k \geq 1\), denote a path in \(G\). Then we define \(f_{\pi} := f_{n_k} \circ f_{n_{k-1}} \circ \ldots \circ f_{n_1}\). Note that the last node, \(n_{k+1}\), has been excluded from the composition. \(f_{\pi}\) specifies the effect of path \(\pi\) in the sense that, for any data flow information \(INF \in L\) entering the path, \(f_{\pi}(INF)\) determines the transformation of \(INF\) when control passes through the nodes of \(\pi\) (excluding the last node).

Example 3.12: An MDS \(\Omega_{RD}\) for the problem of reaching definitions

Such a system has essentially already been defined in Section 3.5.1. We now specify it formally as an MDS:

1. \((L, \wedge) = (\mathcal{P}(S.DEFS), \cup)\)

   We have already seen (Example 3.11) that \((\mathcal{P}(S.DEFS), \cup)\) is a bounded semilattice with 0 and 1, where 0 corresponds to \(S.DEFS\) and 1 to \(\emptyset\).

2. The function specifying the effect of a basic block \(n\) has the form:
   \[ f_n(X) = (X \cap S.PRE(n)) \cup S.DEF(n) \quad \text{for every } X \subseteq S.DEFS \]

   Clearly, \(f_n(X_1 \land X_2) = f_n(X_1) \land f_n(X_2)\); thus \(f_n\) is distributive.

Now let \(F\) be the set of all functions \(f: L \to L\) such that \(f(X) = (X \cap \text{pre}) \cup \text{def}\), where \(\text{pre}\) and \(\text{def}\) are arbitrary elements of \(L\). Then \(F\) is a distributive function space. Note that the definition of \(F\) is independent of any particular program graph.
(3) Let $G$ be the program graph of Example 3.2. From the analysis results given in Example 3.8 we can immediately determine all functions $f_i$ $(1 \leq i \leq 4, X \subseteq S._{DEFS})$:

- $f_1(X) = (X \cap \{S6\}) \cup \{S1,S2,S3,S4\}$
- $f_2(X) = (X \cap \{S1,S2,S4,S8,S9\}) \cup \{S5,S6\}$
- $f_3(X) = (X \cap \{S1,S3,S5\}) \cup \{S8,S9\}$
- $f_4(X) = X$

Consider the path $\pi = (1,2,3,2)$ in $G$. Then $f_\pi = f_3 \circ f_2 \circ f_1$, and $f_\pi(X) = \{S1,S5,S6,S8,S9\}$, independent of $X$.

We can now introduce the concept of an optimal solution for a data flow system $\Omega$. We have assumed $\text{pred}(s) = \emptyset$. Let $\text{null} \in L$ denote the value to be associated with the initial node. In our intraprocedural setting this value will represent ‘no information’, which will correspond to 0 or 1 in the semilattice. For example, $\text{RD}(s)$ must be defined as the empty set, which corresponds to the 1 element of the lattice. Let $\text{PATH}(n)$ for each $n \neq s$ denote the set of all paths from the initial node to $n$. We can now define the optimal solution for node $n$ to be the meet over the effects of all paths reaching $n$. Because of the way in which this is defined, we also speak of the meet over all paths or MOP solution. Formally, this is defined as follows.

**Definition 3.26** Let $\Omega = (L, \land, F,G,FM)$ be an MDS, where the initial node, $s$, of $G$ is associated with the value $\text{null}$, and $\text{pred}(s) = \emptyset$. The optimal solution of $\Omega$ is given by a function $\text{OPT}: N \rightarrow L$, where:

1. $\text{OPT}(s) = \text{null}$
2. For every $n \in N - \{s\}$: $\text{OPT}(n) = \bigwedge_{n \in \text{PATH}(n)} f_\pi(\text{null})$

It can be shown that the problem of determining the optimal solution of an MDS is undecidable (for a proof see Hecht (1977)).

**Theorem 3.3** There is no algorithm that computes $\text{OPT}$ for every monotone data flow system $\Omega$.

Let $X: N \rightarrow L$ denote any total function that associates nodes with lattice elements. We shall define such a function as conservative or safe iff for all
$n \in N$, $X(n) \leq OPT(n)$. A conservative function takes into account the effect of all paths from the initial node to $n$, and thus can safely be used to control optimization transformations. The general iterative algorithm (Algorithm 3.6) determines a conservative approximation to $OPT$, which will be identical to $OPT$ for the subclass of distributive data flow systems.

Let us conclude this subsection with two remarks concerning Definition 3.26. For any node $n \neq s$, the set $PATH(n)$ may be countably infinite. However, the conditions specified in the definition of the MDS, in particular the boundedness of the semilattice and the monotonicity of the functions $f \in F$, guarantee that $OPT(n)$ is defined for every $n$.

Finally, we must ask ourselves whether the use of the term 'optimal' is justified here. It is, on the premise that all paths in the program graph are paths that can actually be taken in each program execution. In a particular execution of a real program it may happen that certain paths of the program graph are excluded. In such a case, $OPT$ specifies a suboptimal, but conservative, solution.

### 3.5.5 The general iterative algorithm

In this section we give the general iterative algorithm, which can be considered to be an adaptation of the fixpoint algorithm (Algorithm 3.5) for MDSs. We have already used a version of this algorithm to solve the reaching definitions problem (Algorithm 3.4). In that case, we did not specify an order for the processing of the nodes in one iteration. Here we impose rPostorder on the nodes of the flowgraph; this will be seen to guarantee the algorithm to be effective.

**Algorithm 3.6: The general iterative algorithm**

**Input** An MDS $\Omega = (L, \wedge, F, G, FM)$ with $G = (N, E, s)$ and $N$ ordered according to rPostorder.

**Output** $INF: N \rightarrow L$, a total function.

**Method**

/* The algorithm uses the variables new, stable and $INF(n)$ for every $n$ with the same meaning as in Algorithm 3.4 */

begin
/* Initialization */

$INF(s) := NULL;$

for every $n \in N$ do $INF(n) := 1$ end for;
/* Iteration */
repeat
  stable := true;
  for every \( n \in N \setminus \{s\} \) in rPostorder do
    new := \( \bigwedge_{n \in \text{pred}(n)} f_{s(n)}(\text{INF}(n')) \);
    if new \( \neq \) \( \text{INF}(n) \)
      then \( \text{INF}(n) := \text{new} \); stable := false
    fi
  end for
until stable
end

The following results concern the behavior and output of the algorithm. The proofs are to be found in Kam and Ullman (1976, 1977) and Hecht (1977).

**Theorem 3.4** Algorithm 3.6 terminates and yields the greatest fixpoint of the following equation system:

1. \( X(s) = \text{NULL} \)
2. \( X(n) := \bigwedge_{n' \in \text{pred}(n)} f_{s(n')}(X(n')) \quad n \in N \setminus \{s\} \)

**Theorem 3.5** For every node \( n \): \( \text{INF}(n) \leq \text{OPT}(n) \).

This theorem implies that the solution is conservative. A consequence of this is that all transformations which are based on \( \text{INF} \) rather than \( \text{OPT} \) are semantically safe.

**Theorem 3.6**

1. If Algorithm 3.6 is applied to a distributive MDS, then \( \text{INF} = \text{OPT} \).
2. Let \((L, \wedge)\) be a bounded semilattice with 0 and 1, and \( F \) a monotone function space for \( L \) that is non-distributive. Then there exist \( G \) and \( FM \) such that the application of Algorithm 3.6 to \( \Omega = (L, \wedge, F, G, FM) \) yields a suboptimal solution: \( \text{INF}(n) < \text{OPT}(n) \) for an \( n \in N \).

In general, the repeat-loop of Algorithm 3.6 needs on the order of \(|N|\) iterations, from which we obtain a worst-case complexity of \( O(|N|^2) \).
operations. In contrast, if Algorithm 3.6 is applied to a reducible flow-graph, then for an important class of data flow systems at most \( d + 2 \) iterations are needed, where \( d \) is the loop-connectedness of the graph (Kam and Ullman, 1976); thus in most cases the algorithm will terminate after at most five iterations. This includes bit vector problems, as discussed below.

### 3.5.6 Problem classes

The iterative algorithm will execute efficiently when the order in which the nodes of the flowgraph are visited in each iteration coincides with the direction in which data flow information is being propagated. The dominance relation provides us with a simple way to capture this notion of order precisely: we say that any linear order of the nodes which topologically sorts dominance is top-down, and the inverse of this is bottom-up. The data flow problems of reaching definitions and available expressions are both top-down problems. The condition imposed on the set \( N \) in the general iterative algorithm implicitly reflects that assumption, as rPostorder topologically sorts the dominance relation.

There are problems in which the natural order of information propagation is bottom-up. An example is the live variables problem (see Sections 3.4.1 and 3.6.2). In such cases, the numbering of the nodes in the iterative algorithm should be inverted.

Note that for a single-exit flowgraph \( G \) a bottom-up problem can be easily converted into a top-down problem for \( G^{-1} \), and vice versa. Thus the two classes of problems are closely related.

There is another, orthogonal, classification which distinguishes between existence problems and all problems. RD belongs to the former category, because it determines whether a path exists along which a given definition can reach a basic block. The same is true for the live variables problem. Conversely, the available expressions problem is of the latter type, as we consider only those expressions that are computed on all paths reaching a node. The difference between these two classes of problems is reflected in the definition of the join operation on a powerset: for existence problems it is the union, otherwise the intersection.

Finally, some data flow problems can be encoded as bit vector problems. Assume that \( L = \mathcal{P}(M) \) for some ‘small’ set \( M \), such as \( S.DEFS \), \( EXP \) or the set of scalar variables. Then the elements of \( L \) can be represented as bit vectors of length \( |M| \), and the meet operation corresponds to a bit vector ‘and’ or ‘or’. The RD, available expressions and live variables problems all belong to this category. These data flow systems can be solved efficiently (see the closing paragraph of Section 3.5.5). A problem that cannot be modeled in this way is constant propagation, which is the subject of Section 3.6.4.
3.6 Examples for data flow systems

Let $\Omega = (L, \land, F, G, FM)$ denote an MDS. In the following we sometimes find it convenient to ignore the graph $G$ and the function $FM$ and consider instead systems $\phi = (L, \land, F)$. These are known as monotone data flow analysis frameworks (MDFs). An MDS can then be interpreted as an instance of an MDF, specified by a triple $(\phi, G, FM)$. Note that our discussion in Section 3.5 was completely independent of a particular choice of $G$ and $FM$. Concepts such as distributivity can be immediately applied to MDFs.

In this section we shall apply the theory developed thus far to four data flow problems: available expressions, live variables, dominance, and constant propagation. For each of these problems, an MDF will be specified. For RD, we have already done so in Example 3.12.

3.6.1 Available expressions

The available expressions problem (AE) is the problem of determining, for each node $n$ of a flowgraph, the set of all available expressions reaching $n$. Remember that expression $exp \in EXP$ is available at $n$ iff every path reaching $n$ evaluates $exp$, and the subpath between the last computation of $exp$ and $n$ is definition-free for all operands of the expression (see Section 3.4.2).

An expression $exp$ is said to be generated in a basic block $n$ if it is evaluated in $n$, and $n$ does not contain a definition of any of its operands after the last evaluation of $exp$. An expression is said to be preserved in $n$ if none of its operands is in $DEF(n)$. These relationships are described by two local sets associated with each basic block.

Definition 3.27 Let $n$ denote an arbitrary basic block.

1. $GEN\_EXP(n) := \{exp \in EXP: exp$ is generated in $n\}$
2. $PRE\_EXP(n) := \{exp \in EXP: exp$ is preserved in $n\}$.

The MDF $\phi_{AE} = (L, \land, F)$ for AE can be specified as follows:

1. $(L, \land) = (\Phi(EXP), \land)$

This is a bounded semilattice where $0 = \emptyset$, $1 = EXP$, and $\leq$ corresponds to set inclusion.
(2) The effect of a basic block \( n \) can be described by a function \( f_n \) as follows:

\[
f_n(X) = (X \cap \text{PRE.EXP}(n)) \cup \text{GEN.EXP}(n) \quad \text{for every } X \subseteq \text{EXP}.
\]

All functions \( f_n \) are distributive. Now let \( F \) be the set of all functions \( f : L \rightarrow L \) such that \( f(X) = (X \cap \text{pre}) \cup \text{gen} \), where \( \text{pre} \) and \( \text{gen} \) are arbitrary elements of \( L \). \( F \) is a distributive function space.

\( \phi_{\text{AE}} \) is a distributive MDF.

### 3.6.2 Live variables

The live variables problem (LV) is the problem of determining, for each node \( n \) of a flowgraph, the set of all variables that are live at the end of \( n \). Remember that \( v \) is live at the exit of \( n \) iff there is a path from \( n \) to \( n' \) such that there is an outward exposed use of \( v \) in \( n' \), and the path, except for its endpoints, is definition-free for \( v \).

LV is a bottom-up problem: data flow information is attached to the end of basic blocks and propagated through the flowgraph in an order that topologically sorts the inverse of the dominance relation. The functions \( f_n \) associated with basic blocks describe the transformation which data flow information undergoes when it is passed from the exit to the entry of the block. We assume a single-exit flowgraph. The general iterative algorithm must be modified in three places. Firstly, the for-loop is processed in inverse rPostorder. Secondly, in the assignment to the variable new, the meet is taken over all \( n' \in \text{succ}(n) \). Third, the exit node of the flowgraph (rather than the initial node) is initialized to \( \text{NULL} \). Everything else remains the same.

The MDF \( \phi_{\text{LV}} = (L, \wedge, F) \) for LV can be specified as follows:

(1) \[
(L, \wedge) = (\emptyset(\text{VAR}), \cup)
\]

This is a bounded semilattice where \( 0 = \text{VAR}, 1 = \emptyset, \) and \( \leq \) corresponds to the inverse of set inclusion.

(2) The effect of a basic block \( n \) can be described by a function \( f_n \) as follows:

\[
f_n(X) = (X \cap \text{PRE}(n)) \cup \text{USE}(n) \quad \text{for every } X \subseteq \text{VAR}
\]

All functions \( f_n \) are distributive, and a distributive function space \( F \) can be constructed just as for the other frameworks.

\( \phi_{\text{LV}} \) is a distributive MDF.
3.6.3 Dominance

Let $G = (N, E, s)$ be a flowgraph. Then the definition of dominance (Definition 3.7) implies

$$\text{DOM}(n) = \bigcap_{n' \in \text{PATH}(n)} \{ n' \in N : n' \text{ is a node in PATH}(n) \}.$$

From this, we construct the following MDS:

1. $(L, \wedge) = (\mathcal{P}(N), \cap)$

   This is a bounded semilattice where $0 = \emptyset$, $1 = N$, and $\leq$ corresponds to set inclusion.

2. The effect of basic block $n$ can be modeled as follows:

   $$f_n(X) = X \cup \{n\} \text{ for every } X \subseteq N$$

   The function $f_n$ models the effect of basic block $n$ and is distributive for each $n$. We can construct a distributive function space $F$ satisfying Definition 3.10 based on the $f_n$.

The data flow system $\Omega_{\text{DOM}} = (L, \wedge, F, G, FM)$ is distributive, and $\text{OPT}(n) = \text{DOM}(n) - \{n\}$ for every $n \in N$.

3.6.4 Constant propagation

Constant propagation (CP) can be formulated as the problem of determining the value of a variable at a given point of the program whenever this value is constant. We shall model this problem by an MDF $\phi_{\text{CP}}$; it turns out that $\phi_{\text{CP}}$ is non-distributive, which means that only an approximation can be computed by the iterative algorithm.

When propagating data flow information for CP, for each variable $v \in \text{VAR}$

1. the value of $v$ is a known constant, $c$, or
2. the value of $v$ is not constant, or
3. it is not known whether (1) or (2) applies.

Let $\Gamma' := \Gamma \cup \{\text{nonconst}\}$, where $\Gamma$ is the universal domain introduced in Section 3.2. The 'artificial values' nonconst and undef are used to model cases (2) and (3), respectively. For the remainder of this section, the concept of 'value' will refer to any element of $\Gamma'$. Remember we have defined $\Gamma_r := \Gamma(\text{undef})$ as the set of proper values.
Lemma 3.5  Let the binary operation \( \wedge \) in \( \Gamma' \) be defined as follows:

1. For all \( x \in \Gamma' \): \( x \wedge \text{nonconst} := \text{nonconst} \).
2. For all \( x \in \Gamma' \): \( x \wedge \text{undef} := x \).
3. For all proper values \( c, d \in \Gamma' \):
   \[
   c \wedge d := \begin{cases} 
   c & \text{if } c = d \\
   \text{nonconst} & \text{if } c \neq d 
   \end{cases}
   \]

Then \( (\Gamma', \wedge) \) is a bounded semilattice with \( 0 = \text{nonconst} \) and \( 1 = \text{undef} \). The length of every chain is at most 2.

Proof  The operation \( \wedge \) is idempotent, commutative and associative; thus \( (\Gamma', \wedge) \) is a semilattice. From (1) and (2) we see that \( \text{nonconst} \) and \( \text{undef} \), respectively, are the 0 and 1 elements of the semilattice. The associated partial order \( \leq \) is defined by:

4. \( \text{nonconst} \leq x \) for all \( x \in \Gamma' \).
5. \( x \leq \text{undef} \) for all \( x \in \Gamma' \).
6. For all \( c, d \in \Gamma' \), with \( c \neq d \): there is no relationship between \( c \) and \( d \).

This concludes the proof. A lattice of this type is called flat; it can be represented by a Venn diagram as shown in Figure 3.13.

We are now in a position to construct the lattice needed for the constant propagation problem. Let the notion of state, as introduced in Definition 3.5, be extended to include \( \text{nonconst} \). Then we define \( L \) as the set of all states of \( \text{VAR} \): \( L := \{ \beta : \beta \) is a total function, \( \beta : \text{VAR} \rightarrow \Gamma' \} \). Let \( \beta, \beta' \in L \). The meet operation in \( L \) is defined as follows: for every \( v \in \text{VAR} \), let \( (\beta \wedge \beta')(v) := \beta(v) \wedge \beta'(v) \). It can now easily be shown that \( (L, \wedge) \) is a bounded semilattice with 0 element \( \beta_{\text{zero}} \) and 1 element \( \beta_{\text{one}} \), where for every \( v \in \text{VAR} \):

\[
\beta_{\text{zero}}(v) = \text{nonconst} \quad \text{and} \quad \beta_{\text{one}}(v) = \text{undef}.
\]

We still have to model the effect of statements on elements of \( L \). We do so by handling assignment statements \( S \): \( A = B \sigma C \), where \( A, B, C \in \text{VAR} \) and \( \sigma \) is a binary operator. \( \sigma \) is assumed to be a total function \( \sigma : \Gamma' \times \Gamma' \rightarrow \Gamma' \). Other statements can be treated similarly.

\footnote{This means that we do not consider operators such as \('\)' and strategies for handling statements such as \( A = 1/0 \) at compile time.}
Let $\beta \in L$ be given arbitrarily, and define $\beta' := f_s(\beta)$ as follows:

1. $\beta'(v) = \beta(v)$ for all $v \in \text{VAR} - \{A\}$
2. $\beta'(A) = \begin{cases} \text{nonconst} & \text{if } (\beta(B) = \text{nonconst}) \lor (\beta(C) = \text{nonconst}) \\ \\
\text{undef} & \text{or if } (\beta(B) = \text{undef}) \lor (\beta(C) = \text{undef}) \\ \\
\text{else} & \text{if } (\beta(B), \beta(C) \in \Gamma); \text{ the result of } \beta(B) \lor \beta(C) \text{ can be computed in the compiler. Let } \text{val} \text{ denote the value yielded by this operation } */ \\
\text{val} & \end{cases}$

It remains to be shown that $f_s$ is monotonic, that is

$$\beta_1 \leq \beta_2 \implies f_s(\beta_1) \leq f_s(\beta_2)$$

We must only consider the case for variable $A$, that is we must verify

$$\beta_1 \leq \beta_2 \implies f_s(\beta_1(A)) \leq f_s(\beta_2(A))$$

We see that this holds by considering all possible values for $B$ and $C$ in $\beta_1$ and $\beta_2$ and taking into account the fact that $\beta_1(X) < \beta_2(X)$ will only be true, for any variable $X$, if:

$$\beta_1(X) = \text{undef} \text{ and } \beta_1(X) \in \Gamma; \text{ or } \beta_1(X) = \text{nonconst}, \text{ or } \beta_2(X) = c \in \Gamma; \text{ and } \beta_1(X) = \text{nonconst}$$

Thus a monotone function space $F$ can be associated with $L$ and we have modeled constant propagation as a data flow system.

However, this system is not distributive: if it were, then $f(\beta_1 \land \beta_2) = f(\beta_1) \land f(\beta_2)$ would have to hold for all pairs $\beta_1, \beta_2 \in L$. It therefore suffices to construct a counterexample. Let $\text{VAR} = \{A, B, C\}$ be linearly ordered, in the order of writing, $\beta_1 = (2, 3, \text{undef})$, $\beta_2 = (3, 2, \text{undef})$ and consider the function $f$ associated with the assignment $\Gamma = A \lor B$. Then

$$f(\beta_1 \land \beta_2) = f(\text{nonconst}, \text{nonconst}, \text{undef}) = (\text{nonconst}, \text{nonconst}, \text{nonconst})$$
but

\[ f(\beta_1) \land f(\beta_2) = f(2,3,\text{undef}) \land f(3,2,\text{undef}) = (2,3,5) \land (3,2,5) = (\text{nonconst,nonconst,5}). \]

### 3.7 Interprocedural analysis

#### 3.7.1 Introduction

The purpose of interprocedural analysis is to increase the precision of data flow and dependence analysis by determining the relevant effects of procedure calls. In the absence of interprocedural analysis, worst-case assumptions have to be made.

Interprocedural analysis must begin with the construction of the call graph, which is a directed graph representing the calling relationships between the procedures of the program. If formal procedure parameters are used in a program, the call graph can only be built after all possible bindings for such parameters have been found. An algorithm that performs this task is given in Section 3.7.2.

In Section 3.7.3, we discuss in-line expansion, a technique which textually replaces procedure calls by the appropriately modified procedure bodies. Whenever feasible, standard intraprocedural techniques can be applied to the resulting program.

This is followed by a description of several interprocedural data flow problems and solution approaches (Section 3.7.4). Depending on whether or not the set of all control paths inside the procedure is taken into account, problems are respectively classified as flow-sensitive (MUST problems) or flow-insensitive (MAY problems). Some problems exist in both versions: an example is the set of all global variables and formal reference parameters of a procedure \( p \) that MUST/MAY be modified in every activation of \( p \). We emphasize flow-insensitive problems, which are discussed in Section 3.7.4.1; flow-sensitive problems are touched on only briefly in Section 3.7.4.2. Aliasing, the dynamical equivalencing of variables as a result of the argument bindings of formal reference parameters, is then discussed in Section 3.7.4.3. The results of the alias-free analysis and the independent alias analysis can be combined to produce a general solution of a data flow problem.

We base our discussion on the program model introduced in Section 3.2. Only proper procedures (represented by single-entry single-exit flowgraphs) and simple variables will be considered. We further assume that the source code of all procedures is completely accessible for analysis. Remember that the parameter vector of a procedure \( p \), that is, the list of formal parameters in the order of their occurrence in the procedure's declaration, is denoted by \( \nu_p \).