1

An Introduction to Data Flow Analysis

Data flow analysis is a process of deriving information about the run time behaviour of a program.

This chapter introduces the basic concepts of data flow analysis through a contemporary optimization. Then we describe common properties of program analyses at an abstract level and instantiate them for data flow analysis.

1.1 A Motivating Example

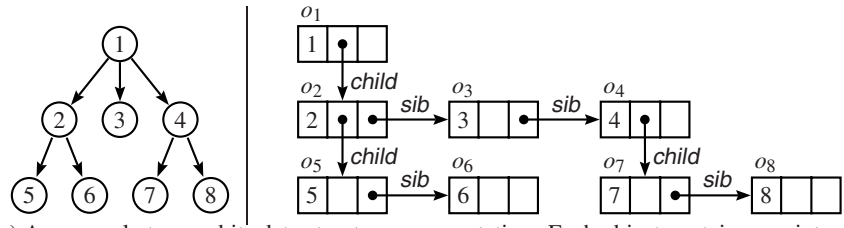
We present a data flow analysis for optimizing heap memory usage in programs to free heap cells as soon as possible. Formal details of the analysis are postponed to Section 4.4. In this section we perform the required analysis and explain the issues involved intuitively. The result of intraprocedural data flow analysis of this program using the formal theory is presented in Section 4.4.5 whereas Section 9.5 presents the result of interprocedural data flow analysis.

1.1.1 Optimizing for Heap Memory

Figure 1.1(b) provides a program to traverse a tree in depth first order. The data structure used for representing the input tree is illustrated in Figure 1.1(a). Function `dfTraverse` recursively descends down a tree node and prints node numbers while unwinding from recursion. Figure 1.1(c) provides its *control flow graph*. The nodes in this graph represent statements and the edges represent control transfers between the statements. Observe that the `while` loop, which is a compound statement, has been translated in terms of a conditional branch (out edges of block n_2) and an unconditional branch (out edge of block n_5).

For simplicity of descriptions, we assume that reading a pointer is equivalent to reading the data pointed to by the pointer. Further, when we say that a given data object is read, we mean that *some* pointer which points to the data object is read; when a data object is not read, *no* pointer which points to the data object is read.

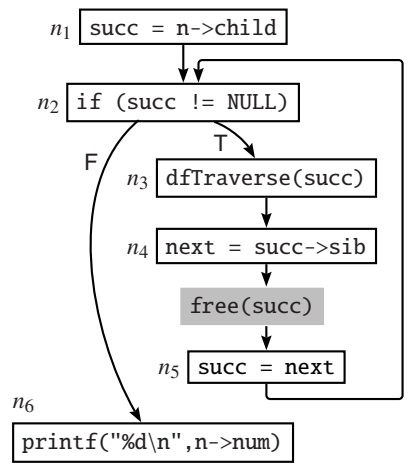
Figure 1.2 provides the execution trace of `dfTraverse` on the input tree in Figure 1.1(a). It is clear from the trace that the data object pointed to by pointer `succ` is last read in block n_4. Thus it is desirable that the heap memory allocated for this



(a) An example tree and its data structure representation. Each object contains a pointer to its first child. Other children are siblings of the first child.

```
 0.  void main()
 1.  {  Tree *tree;
 2.     tree = createTree();
 3.     dfTraverse(tree);
 4.  }
 5.  void dfTraverse(Tree *n)
 6.  {  Tree *succ, *next;
 7.     succ = n->child;
 8.     while (succ != NULL);
 9.     {  dfTraverse(succ);
10.        next = succ->sib;
           free(succ);
11.        succ = next;
12.    }
13.    printf("%d\n",n->num);
14. }
```

(b) A tree traversal program



(c) CFG of `dfTraverse`

FIGURE 1.1
An example of heap memory optimization. Various nodes of `tree` are freed as shown in the gray boxes as soon as their traversal is over.

data object be reclaimed as soon as possible and added to the free pool for a possible subsequent allocation. The statement which performs the suggested deallocation has been shown in gray box and is not part of the original program. Observe that this deallocation cannot be performed through garbage collection because all these data objects are reachable from the root variable `tree` in the program.

This particular instance of optimization can be summarized as follows:

> Pointer variable `succ` is not live at the entry of n_5 and is not aliased to any live pointer. Hence the data can be deallocated at the entry of n_5.

The properties of *liveness* and *aliasing* of pointers are defined as:

Liveness of a pointer. A pointer is live at a program point u if the address that it holds at u is read along some path starting at u.

Aliasing of pointers. Two pointers are aliased to each other at a program point u if they hold the same address in some execution instance of u.

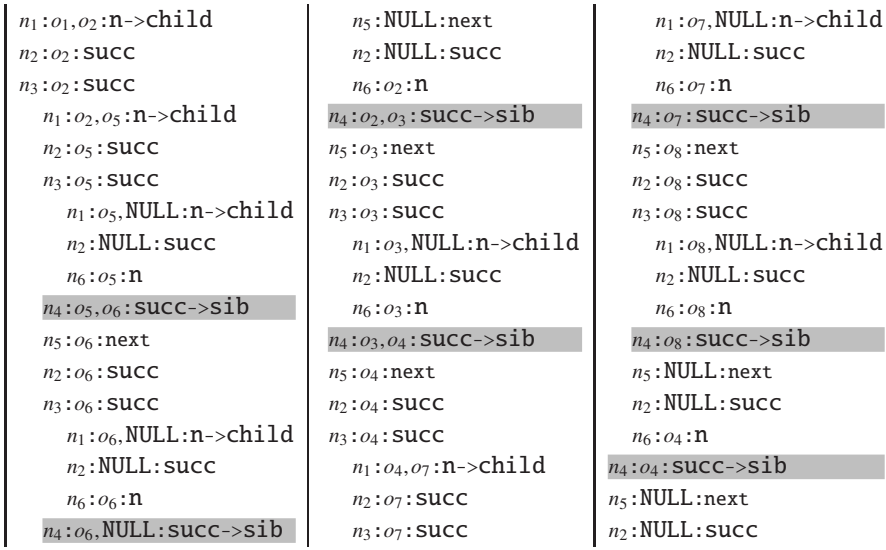


FIGURE 1.2

Execution trace of function `dfTraverse` on the input tree in Figure 1.1(a). Each entry is of the form $x : y : z$ where y is the list of objects read using the pointer sequence z in block x. Entries with gray background correspond to the last use of the first object in the list. Nested activations have been shown by nested indentations.

The final data flow information which enables this optimization has been provided in Figure 1.4 (Section 1.1.4).

The liveness and alias analyses required for performing optimization such as above use the concept of an *access path* which is a sequence of pointers representing a path in the memory. The first pointer in the sequence is a local or global variable whereas all subsequent pointers are field members of structures. In our example, when `succ` points to object o_2, objects o_3, o_4, o_5, o_6 can be accessed using access paths *succ*→*sib*, *succ*→*sib*→*sib*, *succ*→*child*, and *succ*→*child*→*sib*; we say that objects o_3, o_4, o_5, and o_6 are *targets* of access paths *succ*→*sib*, *succ*→*sib*→*sib*, *succ*→*child*, and *succ*→*child*→*sib* respectively.

For the purpose of this chapter, we do not distinguish between access paths beyond two levels of pointer indirections. Access paths with three or more pointers are summarized by suffixing a ⋆ after the first two pointers. Thus *succ*→*sib*→*sib* and *succ*→*sib*→*child* are both represented by *succ*→*sib*→⋆. A more precise and formal method of summarization of access paths using graphs is presented in Section 4.4.3.

Our analyses extend the concept of liveness and aliasing of pointer variables to liveness and aliasing of access paths.

1.1.2 Computing Liveness

The liveness information at a program point is represented by a set of live access paths where liveness of an access path is defined as follows:

Liveness of access paths. An access path ρ is live at a program point u if the targets of all prefixes of ρ are read along some control flow path starting at u.

Clearly, liveness sets are prefix-closed. For notational convenience, we retain only those access paths which are not prefixes of other access paths.

Since liveness information at u represents possible uses beyond u, it is computed from the liveness information at the successors of u. For an access path to be live at u, it is sufficient that it is live at any successor of u. Hence the set of live access paths at u is a union of the corresponding sets at successors of u. In our example, the liveness set at the exit of n_2 is computed by taking a union of the sets of live access paths at the entries of n_6 and n_3.

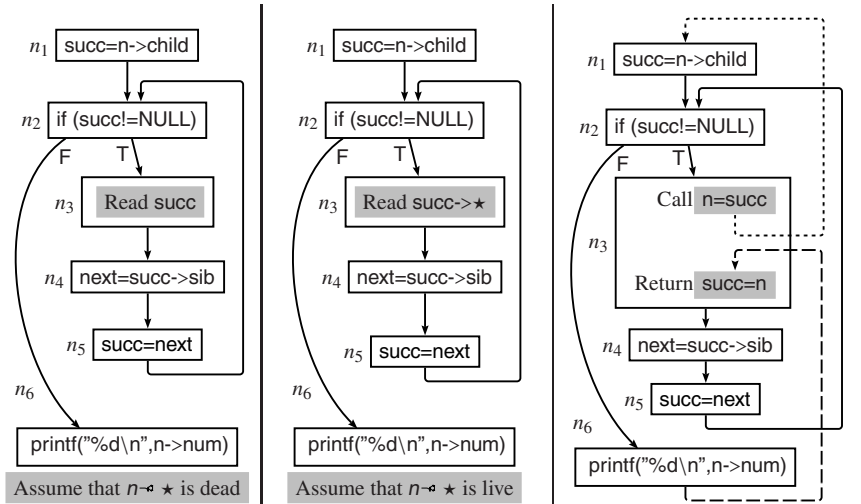
The sets of live access paths are computed by successive refinements starting from a conservative initial value of $\emptyset$. The initial value chosen is $\emptyset$ because it is the identity of union operation. We choose an iterative traversal over the CFG for each step of refinement. Since liveness at a program point depends on the successor points, we traverse the CFG against the direction of control flow. For our example, this implies the following order of computing liveness sets: n_6, n_5, n_4, n_3, n_2, and n_1. This method is called the *round-robin iterative* method of performing data flow analysis. We will use this method in the rest of the book to present our examples. Sections 3.4, 3.5, and 5.2 define this method formally and analyze its complexity.

Modelling Interprocedural Effects

The data flow information within a function is influenced by interprocedural effects arising out of function calls. In particular, the data flow information in function f is influenced by the caller functions of f as well as by the functions called by f. If the interprocedural effects are ignored during intraprocedural analysis, it could lead to incorrect results. This can be avoided by either performing interprocedural analysis or by approximating the interprocedural effects.

Figure 1.3 models the above situations for our example program. Figure 1.3(a) illustrates the situation when the interprocedural effects are ignored: The call statement in block n_3 is modeled as reading merely the actual parameter `succ`. Further it is assumed that no access path rooted at the formal parameter `n` is live at the exit of `dfTraverse`. Figure 1.3(b) shows a safe approximation of liveness for handling interprocedural effects: In block n_3, it is assumed that any access path rooted at the actual parameter `succ` becomes live due to the call made in n_3. Similarly, it is assumed that any path rooted at the formal parameter `n` is live at end of `dfTraverse` because it may be accessed in a caller's body using the actual parameter.

Figure 1.3(c) shows how the function `dfTraverse` can be represented to facilitate interprocedural analysis. It models function calls by splitting them into a call node and a return node and by adding an edge from the call node to the start of the



(a) Intraprocedural analysis ignoring interprocedural effects.

(b) Intraprocedural analysis with interprocedural approximation.

(c) Interprocedural analysis. Function main has not been shown.

FIGURE 1.3

Modelling interprocedural effects in liveness analysis for the program in Figure 1.1.

called procedure and an edge from the end of the called procedure to the return node. A call node maps the actual parameters to the formal parameters. During liveness analysis, the call node in block n_3 transfers the liveness of the formal parameter `n` in the callee's body (`dfTraverse`) to the liveness of the actual parameter `succ` in the caller's body (also `dfTraverse`). In our example, the callee does not return any value. However, since the parameter of `dfTraverse` is a pointer variable, the return node in block n_3 transfers the liveness of the actual parameter `succ` in the caller's body to the liveness of the formal parameter `n` in the callee's body.

For simplicity of exposition, we first show the liveness analysis for simple intraprocedural analysis (modeled in Figure 1.3(a)). Then we show the effect of incorporating the interprocedural approximation (modeled in Figure 1.3(b)). Finally we show a simple interprocedural liveness analysis (modeled in Figure 1.3(c)).

In the later part of the book, a solution of the simple intraprocedural liveness analysis of our example program as well as intraprocedural liveness analysis with interprocedural summarization has been presented in Section 4.4.5. Common variants of interprocedural data flow analysis are later introduced in Section 7.6 and Section 9.5 presents interprocedural liveness analysis of our example.

Simple Intraprocedural Liveness Analysis

As described before, simple intraprocedural analysis disregards the interprocedural effects completely. Thus it is assumed that no access path is live at the end of the procedure. Liveness information at the end of the first iteration is:

Block	Liveness at Exit	Liveness at Entry	Remark
n_6	$\emptyset$	{*n*}	Liveness of *n* is generated.
n_5	$\emptyset$	{*next*}	Liveness of *next* is generated.
n_4	{*next*}	{*succ*→▷*sib*}	Liveness of *next* is killed. Liveness of *succ*→▷*sib* is generated.
n_3	{*succ*→▷*sib*}	{*succ*→▷*sib*}	Liveness of *succ* is generated. Liveness of *succ*→▷*sib* is propagated.
n_2	{*n*,*succ*→▷*sib*}	{*n*,*succ*→▷*sib*}	Liveness is propagated.
n_1	{*n*,*succ*→▷*sib*}	{*n*→▷*child*→▷⋆}	Liveness of *succ*→▷*sib* is transferred to `n` and is summarized.

Liveness computation in block n_1 illustrates the process of transferring liveness from one access path to the other access paths. The target objects of *succ* at the exit of n_1 are target objects of *n*→▷*child* at the entry of n_1. Hence the live access path *succ*→▷*sib* from the exit of n_1 is transferred to the entry of n_1 as *n*→▷*child*→▷*sib* which is then summarized to *n*→▷*child*→▷⋆; this also subsumes the unchanging live access path *n*. The process of transfer is described as follows:

> If access path $a \triangleright \sigma$ is live after an assignment `a = b`, then σ is transferred to b and the access path $b \triangleright \sigma$ becomes live before the assignment.

Data flow information converges in the third iteration as shown below. In the second iteration, liveness information {*n*,*succ*→▷*sib*} at the entry of n_2 is propagated to the exit of n_5 along the back edge. The assignment in n_5 does not affect *n*, but the access paths *succ*→▷*sib* cease to be live before n_5 due to the assignment to `succ` and the liveness of *succ*→▷*sib* is transferred as the liveness of *next*→▷*sib* before the assignment. Computing liveness in n_4 involves transfer followed by summarization.

Block	Liveness in iteration 2		Liveness in iteration 3	
	At Exit	At Entry	At Exit	At Entry
n_6	$\emptyset$	{*n*}	$\emptyset$	{*n*}
n_5	{*n*,*succ*→▷*sib*}	{*n*,*next*→▷*sib*}	{*n*,*succ*→▷*sib*→▷⋆}	{*n*,*next*→▷*sib*⋆}
n_4	{*n*,*next*→▷*sib*}	{*n*,*succ*→▷*sib*→▷⋆}	{*n*,*next*→▷*sib*⋆}	{*n*,*succ*→▷*sib*→▷⋆}
n_3	{*n*,*succ*→▷*sib*→▷⋆}	{*n*,*succ*→▷*sib*→▷⋆}	{*n*,*succ*→▷*sib*→▷⋆}	{*n*,*succ*→▷*sib*→▷⋆}
n_2	{*n*,*succ*→▷*sib*→▷⋆}	{*n*,*succ*→▷*sib*→▷⋆}	{*n*,*succ*→▷*sib*→▷⋆}	{*n*,*succ*→▷*sib*→▷⋆}
n_1	{*n*,*succ*→▷*sib*→▷⋆}	{*n*→▷*child*→▷⋆}	{*n*,*succ*→▷*sib*→▷⋆}	{*n*→▷*child*→▷⋆}

Intraprocedural Analysis with Interprocedural Approximation

Interprocedural approximation assumes that *n*→▷⋆ is live at the end of `dfTraverse` and *succ*→▷⋆ is live just before the recursive call. Due to this approximation, the analysis terminates in two iterations.

Block	Liveness in iteration 2		Liveness in iteration 3	
	At Exit	At Entry	At Exit	At Entry
n_6	$\{n\triangleright\star\}$	$\{n\triangleright\star\}$	$\{n\triangleright\star\}$	$\{n\triangleright\star\}$
n_5	$\emptyset$	$\{next\}$	$\{n\triangleright\star, succ\triangleright\star\}$	$\{n\triangleright\star, next\triangleright\star\}$
n_4	$\{next\}$	$\{succ\triangleright sib\}$	$\{n\triangleright\star, next\triangleright\star\}$	$\{n\triangleright\star, succ\triangleright\star\}$
n_3	$\{succ\triangleright sib\}$	$\{succ\triangleright\star\}$	$\{n\triangleright\star, succ\triangleright\star\}$	$\{n\triangleright\star, succ\triangleright\star\}$
n_2	$\{n\triangleright\star, succ\triangleright\star\}$	$\{n\triangleright\star, succ\triangleright\star\}$	$\{n\triangleright\star, succ\triangleright\star\}$	$\{n\triangleright\star, succ\triangleright\star\}$
n_1	$\{n\triangleright\star, succ\triangleright\star\}$	$\{n\triangleright\star\}$	$\{n\triangleright\star, succ\triangleright\star\}$	$\{n\triangleright\star\}$

Interprocedural Analysis

For interprocedural analysis, we split block n_3 into a call block n_3.call and a return block n_3.ret and compute liveness at the entries and exits of these blocks. The initial value is $\emptyset$. No access path is live after the call to `dfTraverse` in function `main`. The liveness information after first two iterations is:

Block	Liveness in iteration 2		Liveness in iteration 3	
	At Exit	At Entry	At Exit	At Entry
n_6	$\emptyset$	$\{n\}$	$\{n\triangleright sib\}$	$\{n\triangleright sib\}$
n_5	$\emptyset$	$\{next\}$	$\{n, succ\}$	$\{n, next\}$
n_4	$\{next\}$	$\{succ\triangleright sib\}$	$\{n, next\}$	$\{n, succ\triangleright sib\}$
n_3.ret	$\{succ\triangleright sib\}$	$\{n\triangleright sib\}$	$\{n, succ\triangleright sib\}$	$\{n\triangleright sib\}$
n_3.call	$\emptyset$	$\emptyset$	$\{n\triangleright child\}$	$\{succ\triangleright child\}$
n_2	$\{n\}$	$\{n, succ\}$	$\{n\triangleright sib, succ\triangleright child\}$	$\{n\triangleright sib, succ\triangleright child\}$
n_1	$\{n, succ\}$	$\{n\triangleright child\}$	$\{n\triangleright sib, succ\triangleright child\}$	$\{n\triangleright sib, n\triangleright child\triangleright\star\}$

In the second iteration, $\{n\triangleright sib\}$ is propagated from the entry of n_3.ret to the exit of n_6 and $\{n\triangleright child\}$ is propagated from the entry of n_1 to the exit of n_3.call. Further, the transfer in block n_1 causes summarization in the second iteration.

	Block	At Exit	At Entry
Iteration 3	n_6	$\{n\triangleright sib\}$	$\{n\triangleright sib\}$
	n_5	$\{n\triangleright sib, succ\triangleright child\}$	$\{n\triangleright sib, next\triangleright child\}$
	n_4	$\{n\triangleright sib, next\triangleright child\}$	$\{n\triangleright sib, succ\triangleright sib\triangleright\star\}$
	n_3.ret	$\{n\triangleright sib, succ\triangleright sib\triangleright\star\}$	$\{n\triangleright sib\triangleright\star\}$
	n_3.call	$\{n\triangleright sib, n\triangleright child\triangleright\star\}$	$\{succ\triangleright sib, succ\triangleright child\triangleright\star\}$
	n_2	$\{n\triangleright sib, succ\triangleright sib, succ\triangleright child\triangleright\star\}$	$\{n\triangleright sib, succ\triangleright sib, succ\triangleright child\triangleright\star\}$
	n_1	$\{n\triangleright sib, succ\triangleright sib, succ\triangleright child\triangleright\star\}$	$\{n\triangleright sib, n\triangleright child\triangleright\star\}$
Iteration 4	n_6	$\{n\triangleright sib\triangleright\star\}$	$\{n\triangleright sib\triangleright\star\}$
	n_5	$\{n\triangleright sib, succ\triangleright sib, succ\triangleright child\triangleright\star\}$	$\{n\triangleright sib, next\triangleright sib, next\triangleright child\triangleright\star\}$
	n_4	$\{n\triangleright sib, next\triangleright sib, next\triangleright child\triangleright\star\}$	$\{n\triangleright sib, succ\triangleright sib\triangleright\star\}$
	n_3.ret	$\{n\triangleright sib, succ\triangleright sib\triangleright\star\}$	$\{n\triangleright sib\triangleright\star\}$
	n_3.call	$\{n\triangleright sib, n\triangleright child\triangleright\star\}$	$\{succ\triangleright sib, succ\triangleright child\triangleright\star\}$
	n_2	$\{n\triangleright sib\triangleright\star, succ\triangleright sib, succ\triangleright child\triangleright\star\}$	$\{n\triangleright sib\triangleright\star, succ\triangleright sib, succ\triangleright child\triangleright\star\}$
	n_1	$\{n\triangleright sib\triangleright\star, succ\triangleright sib, succ\triangleright child\triangleright\star\}$	$\{n\triangleright\star\}$

	Block	At Exit	At Entry
Iteration 5	n_6	{n▷sib▷★}	{n▷sib▷★}
	n_5	{n▷sib▷★,$succ$▷sib, $succ$▷$child$▷★}	{n▷sib▷★,$next$▷sib, $next$▷$child$▷★}
	n_4	{n▷sib▷★,$next$▷sib,$next$▷$child$▷★}	{n▷sib▷★,$succ$▷sib▷★}
	n_3.ret	{n▷sib▷★,$succ$▷sib▷★}	{n▷sib▷★}
	n_3.call	{n▷★}	{$succ$▷★}
	n_2	{n▷sib▷★,$succ$▷★}	{n▷sib▷★,$succ$▷★}
	n_1	{n▷sib▷★,$succ$▷★}	{n▷★}
Iteration 6	n_6	{n▷sib▷★}	{n▷sib▷★}
	n_5	{ n▷sib▷★,$succ$▷★}	{ n▷sib▷★,$next$▷★}
	n_4	{ n▷sib▷★,$next$▷★}	{ n▷sib▷★,$succ$▷sib▷★}
	n_3.ret	{ n▷sib▷★,$succ$▷sib▷★}	{n▷sib▷★}
	n_3.call	{n▷★}	{$succ$▷★}
	n_2	{ n▷sib▷★,$succ$▷★}	{ n▷sib▷★,$succ$▷★}
	n_1	{ n▷sib▷★,$succ$▷★}	{n▷★}

It can be verified that the seventh iteration results in the same liveness at each program point indicating convergence.

A Comparison of Liveness Computed by Three Methods

We reproduce below the liveness information computed by the three methods.

Program Point		Intraprocedural Analysis		Interprocedural Analysis
		Simple	Interprocedural Approximation	
n_6	Exit	∅	∅	{n▷sib▷★}
	Entry	{n}	{n▷★}	{n▷sib▷★}
n_5	Exit	{n, $succ$▷sib▷★}	{n▷★, $succ$▷★}	{ n▷sib▷★, $succ$▷★}
	Entry	{n, $next$▷sib★}	{n▷★, $next$▷★}	{ n▷sib▷★, $next$▷★}
n_4	Exit	{n, $next$▷sib★}	{n▷★, $next$▷★}	{ n▷sib▷★, $next$▷★}
	Entry	{n, $succ$▷sib▷★}	{n▷★, $succ$▷★}	{ n▷sib▷★,$succ$▷sib▷★}
n_3	Exit	{n, $succ$▷sib▷★}	{n▷★, $succ$▷★}	{ n▷sib▷★,$succ$▷sib▷★}
	Entry	{n, $succ$▷sib▷★}	{n▷★, $succ$▷★}	{$succ$▷★}
n_2	Exit	{n, $succ$▷sib▷★}	{n▷★, $succ$▷★}	{ n▷sib▷★, $succ$▷★}
	Entry	{n, $succ$▷sib▷★}	{n▷★, $succ$▷★}	{ n▷sib▷★, $succ$▷★}
n_1	Exit	{n, $succ$▷sib▷★}	{n▷★, $succ$▷★}	{ n▷sib▷★, $succ$▷★}
	Entry	{n▷$child$▷★}	{n▷★}	{n▷★}

It is easy to see that the simple intraprocedural analysis fails to record some access paths as live. For example, access path n▷sib is live at the end of `dfTraverse`. This is because the procedure traverses the next sibling of `n` after traversing `n`. However, the simple intraprocedural analysis concludes that it is not live. When interprocedural summarization is included, it records n▷sib as live at the end of the procedure but

it also marks $n{\triangleright}child$ as live. The interprocedural analysis correctly recognizes that only $n{\triangleright}sib$ is live at the end of the procedure.

1.1.3 Computing Aliases

Computing alias information is simpler compared to liveness for this example because there are no interprocedural effects. This is because unlike liveness which is a property of an access path, aliasing at a program point is a relation between two access paths that are visible at that program point. Since there are no global variables, and no assignments to formal parameter in our example, aliases created in `dfTraverse` are restricted to a single activation.

Aliasing of access paths. Access path ρ_1 and ρ_2 are aliased to each other at a program point u, denoted $\rho_1 \doteq \rho_2$, if their targets are same at u along some control flow path reaching u.

Aliasing information at a program point is represented using a set of alias pairs $\rho_1 \doteq \rho_2$. Since an alias holds at a program point u if it holds along some predecessor of u, we use union to combine sets of alias pairs and use its identity ($\emptyset$) as the initial value. Unlike liveness analysis, aliasing information at a program point u depends on the aliases at predecessors of u. Hence we traverse control flow graph along the control flow for faster convergence of successive refinements. This implies the following order: n_1, n_2, n_3, n_4, n_5, and n_6.

The aliases at the end of first iteration are as shown below:

Block	Aliases at Entry	Aliases at Exit	Remark
n_1	$\emptyset$	$\{succ \doteq n{\triangleright}child\}$	Generation
n_2	$\{succ \doteq n{\triangleright}child\}$	$\{succ \doteq n{\triangleright}child\}$	Propagation
n_3	$\{succ \doteq n{\triangleright}child\}$	$\{succ \doteq n{\triangleright}child\}$	Propagation
n_4	$\{succ \doteq n{\triangleright}child\}$	$\{succ \doteq n{\triangleright}child,$ $next \doteq succ{\triangleright}sib$ $next \doteq n{\triangleright}child{\triangleright}\star\}$	Propagation, transfer and summarization
n_5	$\{succ \doteq n{\triangleright}child,$ $next \doteq succ{\triangleright}sib$ $next \doteq n{\triangleright}child{\triangleright}\star\}$	$\{succ \doteq next,$ $succ \doteq n{\triangleright}child{\triangleright}\star,$ $next \doteq n{\triangleright}child{\triangleright}\star\}$	Generation, killing and transfer
n_6	$\{succ \doteq n{\triangleright}child\}$	$\{succ \doteq n{\triangleright}child\}$	Propagation

Observe the effect of assignment `next = succ->sib` in block n_4 on the aliases at the entry of n_4. Since *succ* is aliased to $n{\triangleright}child$, *next* gets aliased to $n{\triangleright}child{\triangleright}\star$. This is analogous to the transfer in liveness. In n_5, since assignment `succ = next` modifies `succ`, alias $succ \doteq n{\triangleright}child$ ceases to hold at the exit of n_6. Two new aliases $succ \doteq next$ and $succ \doteq n{\triangleright}child{\triangleright}\star$ are created.

The second iteration causes aliases from the exit of n_5 to be propagated to the entry of n_2 and some more aliases to be generated as a consequence of transfer. Since *succ* is aliased to $n{\triangleright}child{\triangleright}\star$ in block n_4, *next* gets aliased to $n{\triangleright}child{\triangleright}\star$.

Block	Aliases at Entry	Aliases at Exit
n_1	$\emptyset$	$\{succ \doteq n{\triangleright}child\}$
n_2	$\{succ \doteq next, succ \doteq n{\triangleright}child{\triangleright}\star$ $next \doteq n{\triangleright}child{\triangleright}\star\}$	$\{succ \doteq next, succ \doteq n{\triangleright}child{\triangleright}\star,$ $next \doteq n{\triangleright}child{\triangleright}\star\}$
n_3	$\{succ \doteq next, succ \doteq n{\triangleright}child{\triangleright}\star,$ $next \doteq n{\triangleright}child{\triangleright}\star\}$	$\{succ \doteq next, succ \doteq n{\triangleright}child{\triangleright}\star,$ $next \doteq n{\triangleright}child{\triangleright}\star\}$
n_4	$\{succ \doteq next, succ \doteq n{\triangleright}child{\triangleright}\star,$ $next \doteq n{\triangleright}child{\triangleright}\star\}$	$\{succ \doteq n{\triangleright}child{\triangleright}\star,$ $next \doteq succ{\triangleright}sib,$ $next \doteq n{\triangleright}child{\triangleright}\star\}$
n_5	$\{succ \doteq n{\triangleright}child{\triangleright}\star, next \doteq succ{\triangleright}sib,$ $next \doteq n{\triangleright}child{\triangleright}\star\}$	$\{succ \doteq next, succ \doteq n{\triangleright}child{\triangleright}\star,$ $next \doteq n{\triangleright}child{\triangleright}\star\}$
n_6	$\{succ \doteq next, succ \doteq n{\triangleright}child{\triangleright}\star,$ $next \doteq n{\triangleright}child{\triangleright}\star\}$	$\{succ \doteq next, succ \doteq n{\triangleright}child{\triangleright}\star,$ $next \doteq n{\triangleright}child{\triangleright}\star\}$

It can be verified that the third iteration does not compute any new aliases.

1.1.4 Performing Optimization

Figure 1.4 summarizes the final data flow information which enables the desired optimization. Access path *succ* is not live at the exit of n_4 and the entry of n_5. Further at these points none of the access paths that it is aliased to are live. Thus the object pointed to by it can be freed. Although *next* is not live in blocks n_2, n_3, and n_4, it is aliased to a live access path and hence its target cannot be freed. An alternative place for deallocating `succ` is block n_6. The difference between the two deallocations is that the former will be performed after a call to `dfTraverse` is over while the latter will be performed just before the end of a call.

1.1.5 General Observations

At the entry of n_5, access path *succ* is not live. It is aliased to $n{\triangleright}child$ which is not live either. If function `main` is modified to access `tree->child` after the call to `dfTraverse` as shown below, then $n{\triangleright}child$ will be live at the exit of n_6.

```
0.  void main()
1.  {  Tree *tree;
2.     tree = createTree();
3.     printEdges(tree);
       printf("%d\n",tree->child->num);
4.  }
```

Since liveness of $n{\triangleright}child$ is not affected by the assignment in n_5, it will be live at the entry of n_5 too. Thus, with this change, *succ* cannot be freed. Interestingly, this change accesses only object o_2 outside of function `dfTraverse` but prohibits freeing any object in `dfTraverse`. This is because the same statements in `dfTraverse` are used to access all objects and unless the code is rewritten to access o_2 and other objects differently, selective freeing is not feasible.

Program Point		Interprocedural Liveness	Aliases
n_1	Entry	$\{n\triangleright\star\}$	$\emptyset$
	Exit	$\{n\triangleright sib\triangleright\star, succ\triangleright\star\}$	$\{succ \doteq n\triangleright child\}$
n_2	Entry	$\{n\triangleright sib\triangleright\star, succ\triangleright\star\}$	$\{succ \doteq next, succ \doteq n\triangleright child\triangleright\star, next \doteq n\triangleright child\triangleright\star\}$
	Exit	$\{n\triangleright sib\triangleright\star, succ\triangleright\star\}$	$\{succ \doteq next, succ \doteq n\triangleright child\triangleright\star, next \doteq n\triangleright child\triangleright\star\}$
n_3	Entry	$\{succ\triangleright\star\}$	$\{succ \doteq next, succ \doteq n\triangleright child\triangleright\star, next \doteq n\triangleright child\triangleright\star\}$
	Exit	$\{n\triangleright sib\triangleright\star, succ\triangleright sib\triangleright\star\}$	$\{succ \doteq next, succ \doteq n\triangleright child\triangleright\star, next \doteq n\triangleright child\triangleright\star\}$
n_4	Entry	$\{n\triangleright sib\triangleright\star, succ\triangleright sib\triangleright\star\}$	$\{succ \doteq next, succ \doteq n\triangleright child\triangleright\star, next \doteq n\triangleright child\triangleright\star\}$
	Exit	$\{n\triangleright sib\triangleright\star, next\triangleright\star\}$	$\{succ \doteq n\triangleright child\triangleright\star, next \doteq succ\triangleright sib, next \doteq n\triangleright child\triangleright\star\}$
n_5	Entry	$\{n\triangleright sib\triangleright\star, next\triangleright\star\}$	$\{succ \doteq n\triangleright child\triangleright\star, next \doteq succ\triangleright sib, next \doteq n\triangleright child\triangleright\star\}$
	Exit	$\{n\triangleright sib\triangleright\star, succ\triangleright\star\}$	$\{succ \doteq next, succ \doteq n\triangleright child\triangleright\star, next \doteq n\triangleright child\triangleright\star\}$
n_6	Entry	$\{n\triangleright sib\triangleright\star\}$	$\{succ \doteq next, succ \doteq n\triangleright child\triangleright\star, next \doteq n\triangleright child\triangleright\star\}$
	Exit	$\{n\triangleright sib\triangleright\star\}$	$\{succ \doteq next, succ \doteq n\triangleright child\triangleright\star, next \doteq n\triangleright child\triangleright\star\}$

FIGURE 1.4
Liveness and alias information in function `dfTraverse`.

This brings out the concept of safety of data flow analysis and the conservative approximations which are used to achieve safety. Since liveness is used to prohibit freeing of objects, it is safer to include spurious access paths as live. Missing a live access path could lead to incorrect optimization. Data flow information is required to represent all possible executions on all possible inputs. Hence the concept of approximation depends on the intended use of the data flow information. Approximations performed by data flow analysis can be characterized by the following two properties: *exhaustiveness* and *safety*. Data flow information is exhaustive if it does not miss any optimization opportunity; it is safe if it does not enable optimizations that do not preserve program semantics. In the context of liveness analysis, exclusion of an access path that is actually live is an approximation towards exhaustiveness because it facilitates freeing a larger number of objects; however, this may be unsafe. In contrast, inclusion of an access path that is not live is an approximation towards safety because it prohibits freeing objects thereby preserving program semantics. The goal of data flow analysis is to compute the most exhaustive safe information.

The interprocedural analysis performed by us is *context insensitive* because it does not distinguish between different calling contexts. In our original example, n becomes live at the exit of `dfTraverse` in those activations of `dfTraverse` that are invoked through the recursive call. It is not live at the end of the outermost activation of `dfTraverse` made through `main`. A *context sensitive* interprocedural analysis can make this distinction. However, exploiting this distinction requires rewriting the code in a non-trivial manner. Otherwise, the data flow information reaching at a program point along different contexts will have to be merged. This highlights the limitation of transformations performed statically. In any case, merging the information discovered by context sensitive analysis generally results in more precise information than the information computed by context insensitive analysis.

The alias analysis performed by us is *flow sensitive* because it propagates aliases along the control flow. A flow insensitive alias analysis disregards the control flow and assumes that the aliases discovered hold at all program points. Such an analysis visits each block only once and accumulates the aliases discovered, no aliases can be killed. For our example, the flow insensitive aliases are: $succ \triangleq n{\triangleright}child{\triangleright}\star$, $succ \triangleq next$, $next \triangleq succ{\triangleright}sib$, and $next \triangleq n{\triangleright}child{\triangleright}\star$. This alias information prohibits freeing the target of $succ$ at the entry of n_5 because it is aliased to $next$ which is live at that point.

We have summarized the access paths n, $n{\triangleright}child$, $n{\triangleright}child{\triangleright}sib$, $n{\triangleright}child{\triangleright}child$, $n{\triangleright}child{\triangleright}sib{\triangleright}sib$, ... by $n{\triangleright}child{\triangleright}\star$. It is clear that some kind of summarization is essential because statically it is not possible to know how many such access paths need to be created by analysis. However for precision, the process of summarization should keep as many access paths distinct in the summary information as is possible. Further, these summaries have to be constructed automatically by data flow analysis. Ensuring convergence on safe summaries requires creating suitable representation for data flow information and devising appropriate operations on the chosen representation. In the case of stack and static data, building summaries is simpler because the mapping between names and addresses does not change during the lifetime of a name and hence names can be directly used to represent data. Section 4.4.3 shows how access paths for heap data can be summarized using graphs.

1.2 Program Analysis: The Larger Perspective

Program analyses cover a large spectrum of motivations, basic principles, and methods. Different approaches to program analysis differ in details but at a conceptual level, almost all program analyses are characterized by some common properties. Although these properties are abstract, they provide useful insights about a particular analysis. A deeper understanding of the analysis would require exploring many more analysis-specific details.

Applications of Analysis

The uses of information derived by program analyses can be broadly classified as:

- *Determining the validity of a program.* An analysis may be used to validate programs with regard to some desired properties (viz. type correctness).
- *Understanding the behaviour of a program.* An analysis may discover useful properties of programs required for debugging, maintenance, verification, or testing etc. Abstract interpretation, slicing, ripple analysis, test data generation etc. are the common examples of such analyses.
- *Transforming a program.* Most analyses enable useful transformations to be performed on programs. Traditionally, the term program analysis has been used for the analyses that facilitate transforming a program within the same given representation. These transformations may be aimed at optimizing the program for space, time, or power consumption. Note that analyses such as lexical and syntax analyses transform a program representation into another representation and are not included in the class of program analyses.
- *Enabling program execution.* Program analysis can also be used for determining the operations implied by a program so that the program can be executed (viz. dynamic type inferencing).

Approaches to Program Analysis

Some of the common paradigms of program analysis are:

- *Inference Systems* consisting of a set of axioms and inductive and compositional definitions constituting rules of inference.

  In such systems, the properties are *inferred* by repeatedly discovering the premises that are satisfied by the program components of interest and by invoking appropriate rules of inference. Note that there is no algorithm that suggests appropriate choice of rules; it is left to the creativity of the user of such a system. As a consequence, such systems may not be decidable.

  Typically, the inference systems are converted to constraint based system (described below) and constraint resolution algorithms are used for inference.

- *Constraint Resolution Systems* consisting of a constraint store and a logic for solving constraints.

  In such systems, a program component *constrains* the semantic properties. These constraints are expressed in form of inequalities and the semantics properties are derived by finding a solution which satisfies all the constraints.

  Often these constraints take advantage of the temporal or spatial structures of data and operations by grouping the related constraints together. Traditionally they have been unconditional, and are called *flow*-based constraints because they have been solved by traversals over trees or general graphs. Grouping of

structured constraints often leads to replacing groups of related inequalities by equations. Structured constraints often lead to more efficient analyses, both in terms of time as well as space.

- *Model Checking* requires creating suitable abstractions of programs as *models* and the desired properties are expressed in terms of boolean formulae. A model checking algorithm then discovers the states in the mode that satisfy the given formulae.

- *Abstract Interpretations* use abstraction functions to map the concrete semantics values to abstract semantics, perform the computations on the abstract semantics, and use concretization functions to map the abstract semantics back to the concrete semantics. The theory of abstract interpretation provides mechanisms to show the soundness of the abstraction functions. The most interesting aspect of this approach is that the algorithms for performing analysis emerge from the construction of abstraction functions.

  This is unlike inference systems, constraints resolution systems, and model checking, where the specifications of analysis are generally based on intuitions of semantics instead of being derived formally from concrete semantics. Hence these three approaches require separate algorithms that perform the specified analyses.

Other approaches like those involving denotational semantics or logic are relatively less common.

In general an analysis can be expressed in any of the above approaches.

Time of Performing Analysis

An analysis performed before the execution of a program is termed *static analysis*, whereas an analysis performed during the execution of a program (in an interleaved fashion) is termed *dynamic analysis*. Thus an interpreter can perform static analysis (by analyzing a program just before execution) as well as dynamic analysis (by analyzing the program during execution). A compiler, however, can perform static analysis only; for dynamic analysis, a compiler must embed extra code in the compiled program as a part of run time support.

In principle, the choice between static and dynamics analysis is governed by the availability of information on which the analysis depends, the amount of precision required and the permissible run time overheads.

An analysis which depends on run time information is inherently dynamic. For example, if type annotations can be omitted in a language and type associations could change at run time, types can be discovered only at run time. This requires dynamic type inferencing. If some amount of imprecision can be tolerated (viz. if precise types are not expected but it is only expected to constrain the set of possible types by ruling out some types before execution), it may be possible to perform an *approximate* static analysis for an otherwise inherently dynamic analysis. This obviates dynamic analysis *only* if a compromise on the precision of information is

acceptable; otherwise it requires a subsequent dynamic analysis. In any case, it reduces the amount of dynamic analysis and hence, run time overheads.

If run time overheads are a matter of concern, dynamic analyses should be either avoided or preceded by corresponding (approximate) static analyses. This often is the case and it should not come as a surprise that, in practice a majority of analyses performed by language processors are indeed static. Besides, many dynamic analyses have a static counterpart. For instance, many languages require array bounds to be checked at run time; optimizing compilers can minimize these checks by a static array bound checking optimization.

Scope of Analysis

Programs can be viewed as hierarchical constructions consisting of structures and sub-structures. Program analyses try to discover information about a program structure by correlating the information discovered for constituent sub-structures. As such, an analysis may be confined to a small sub-structure like an expression, a statement, or to larger sub-structure like a group of statements or function/procedure blocks, or to still larger structures like modules or entire programs. The nature of analysis for the structures and the sub-structures may be different. The sub-structures that belong to the same structure are analyzed independently. Analysis of a structure and its sub-structure may be interleaved or may be non-overlapping (and cascaded); in either case, the larger structure can be analyzed only after their constituent sub-structures. For example, the liveness analysis performed in Section 1.1 requires analysis of basic blocks to discover their effects.

Flow Sensitivity of Analysis

If the information discovered by an analysis at a program point depends on the control flow paths involving the program point and could vary from one program point to another, then the analysis is *flow sensitivity*. Otherwise, it is flow insensitive. Type inferencing in C is flow insensitive whereas that in Ruby is flow sensitive. In general, flow insensitivity is a compromise on precision for achieving efficiency.

Context Sensitivity of Analysis

If the information discovered by an interprocedural analysis for a function could vary from one calling context of the function to another, then the analysis is *context sensitive*. A *context insensitive* analysis does not distinguish between different calling contexts and computes the same information for all calling contexts of a function. Context insensitivity is also a compromise on precision for achieving efficiency.

Granularity of Performing Analysis

An *exhaustive* analysis derives information starting from scratch whereas an *incremental* analysis updates the previously derived information to incorporate the effect of some changes in the programs. These changes may be caused by transformations (typically for optimization) or by user edits (typically in programming environ-

ments). In general, an incremental analysis must be preceded by at least one instance of the corresponding exhaustive analysis.

Program Representations Used for Analysis

An analysis is typically performed on an intermediate representation of the program. Though the theoretical discussions of many analyses are in terms of the source code (viz. in the case of parallelization), in practice these analyses are performed on a suitable internal representation.

These internal representations differ in their "shapes": They may be either linear data structures (viz. a sequence of quadruples), hierarchical data structures (viz. abstract syntax trees), or general non-linear structures (viz. graphs). The graphs may capture linear abstractions of control flow (as in CFGs) or hierarchical abstractions of control flow (as in call graphs).

Single Static Assignment (SSA) form is an interesting representation that does not belong to the above category. SSA form is used for optimization rather than analysis. As a matter of fact, it can be viewed as the result of a different kind of data flow analysis that explicates the data flow information in a CFG.

Representations of Information

Most common representations of information are sets. The elements of these sets may be of states of a model that satisfy given formulae, or program entities that satisfy the given constraints, or facts that hold at a given program point, or trees or graphs representing types. In many cases these elements may be pairs of program entities and the representations of their properties.

Most analyses require these sets to be finite. Some form of summarization may be required if these sets are not finite. Further the representations of individual properties must also be bounded.

1.3 Characteristics of Data Flow Analysis

Data flow analysis statically computes information about the flow of data (i.e., uses and definitions of data) for each program point in the program being analyzed. This information is required to be a safe approximation of the desired properties of the run time behaviour of the program during each possible execution of that program point on all possible inputs.

Data flow analysis is a special case of program analysis and is characterized by the following:

- *Applications*. Data flow analysis can be used for

  - Determining the semantic validity of a program (viz. type correctness based on inferencing, prohibiting the use of uninitialized variables etc.)
  - Understanding the behaviour of a program for debugging, maintenance, verification, or testing.
  - Transforming a program. This is the classical application of data flow analysis and data flow analysis was originally conceived in this context.

- *Approach of Program Analysis*. Data flow analysis uses constraint resolution systems based on equalities. These constraints are often unconditional. The constraints are called Data Flow Equations.

- *Time*. Data flow analysis is mostly static analysis. The Just-In-Time (JIT) compilation and dynamic slicing etc. involve dynamic data flow analysis.

- *Scope*. Data flow analysis may be performed at almost all levels of scope in a program. Traditionally the following terms have been associated with data flow analysis for different scopes in the domain of imperative languages:

  - Across statements but confined to a maximal sequence of statements with no control transfer other than fall through (i.e., within a *basic block*): Local Data Flow Analysis.
  - Across basic blocks but confined to a function/procedure: Global (intraprocedural) Data Flow Analysis.
  - Across functions/procedures: Interprocedural Data Flow Analysis.

  It is also common to use the term local data flow analysis for analysis of a single statement and global data flow analysis for analysis across statements in a function/procedure. Effectively, the basic blocks for such analyses consist of a single statement.

- *Flow Sensitivity*. Data flow analysis is almost always flow sensitive in that it computes point-specific information. In some cases like alias analysis, flow insensitive analyses are also common.

- *Context Sensitivity*. Interprocedural data flow analysis can be context sensitive as well as context insensitive. In general, fully context sensitive analysis is very inefficient and most practical algorithms employ a limited amount of context sensitivity. Context insensitive data flow analysis is also very common.

- *Granularity*. Data flow analysis can have exhaustive as well as incremental versions. Incremental versions of data flow analysis are conceptually more difficult compared to exhaustive data flow analysis.

- *Program Representations*. The possible internal representations for data flow analysis are abstract syntax trees (ASTs), directed acyclic graphs (DAGs), control flow graphs (CFGs), program flow graphs (PFGs), call multigraphs (CGs),

program dependence graphs (PDGs), static single assignment (SSA) forms etc. The most common representations for global data flow analysis are CFGs, PFGs, SSA, and PDGs whereas interprocedural data flow analyses use a combination of CGs (and CFGs or PFGs). Though ASTs can and have been used for data flow analysis, they are not common since they do not exhibit control flow explicitly.

In this book, we restrict ourselves to CFGs and supergraphs created by connecting CFGs of different procedures.

- *Representation of Data Flow Information*. The most common representations are sets of program entities such as variables or expressions satisfying the given property. These sets are implemented using bit vectors. Some analyses use sets of pairs of entities and their properties. For example, constant propagation stores a constantness value for each expression. Some other form of representations such as access paths require summarization.

1.4 Summary and Concluding Remarks

Data flow analysis is a technique of discovering useful information from programs without executing them. This information can be put to a variety of uses. Data flow analysis was conceived in the context of optimization performed by compilers and to date this remains its most dominant application.

Data flow analysis constructs a static summary of the information that represents run time behaviour of a program. Precision of this information depends on the formulation of analysis in terms of the representation of information, rules of summarization, and the algorithms used to compute the information. This chapter has presented a contemporary optimization that demonstrates the importance of these aspects of data flow analysis. We use access paths as a unit of data flow information and summarization is based on treating all access paths beyond two field names as identical.

Our formulation of liveness analysis uses sets of access paths as data flow information; at a given program point, the data flow information depends on the computations that occur after the program point in some execution path. The effect of a statement on the incoming data flow information is incorporated by applying a flow function. In the case of alias analysis, the data flow information is a set of pairs of access paths; at a given program point this information depends on the computations that precede the program point in some execution path. In either case, the data flow information along different paths is combined by taking a union of the sets.

We have also seen that data flow analysis can be restricted to a single procedure by ignoring function calls or can be performed across procedure boundaries. In the latter situation, the calling context of a procedure influences data flow information

and for precision, such an analysis should be context sensitive.

This book builds on the above theme in the following manner:

- Part I presents analysis formulations at the intraprocedural level. This part describes a large number of data flow problems ranging from the classical problems to contemporary problems. It also presents generalizations underlying these problems. In particular, it presents the lattice theoretic modeling of data flow frameworks apart from the generalizations of constant and dependent parts in flow functions and entity functions as constituents of flow functions. It shows how these generalizations lead to tight complexity bounds.

  The final chapter of the first part presents SSA representation of programs which builds an additional layer of abstraction over the control flow graph representation of programs and directly relates the definition points and the use points of data.

- Part II shows how an intraprocedural formulation can be used for interprocedural analysis. The main theme of this part is that the two are orthogonal and hence we avoid methods that are specific to a particular application or a particular data flow framework. This part presents two generic approaches. The first approach is a *functional* approach that constructs context independent summary flow functions of procedures. These flow functions are used at the call points to incorporate the effects of procedure calls. The second approach is a *value-based* approach that computes distinct values for distinct calling contexts; this is achieved by augmenting the data flow values with context information.

- Part III describes the implementation of a GCC based generic data flow analyzer for bot vectors and shows how particular data flow analyses can be implemented by writing simple specifications.

1.5 Bibliographic Notes

Most texts on compilers discuss data flow analysis in varying lengths [3, 10, 40, 75, 76, 105]. Some of them discuss details [3, 10, 76]. An advanced treatment of data flow analysis can be found in the books by Hecht [44], Muchnick and Jones [77], and F. Nielson, H. R. Nielson and Hankin [80].

Historically, the practice of data flow analysis precedes the theory. Hecht [44] reports that the round-robin method of performing data flow analysis can be traced back to Vyssotsky and Wegner [101]. It was an attempt to discover uses of variables that were potentially uninitialized in a Bell Laboratories 7090 Fortran II compiler. This was the first variant of an analysis that later came to be known as *reaching definitions analysis*. We describe this analysis in Chapter 2. A more powerful variant of this analysis considers transitive effects of assignments and is described in Chapter 4.

The problem of early deallocation of heap memory is an important optimization and has been attempted in many different ways. The fact that there is ample scope for performing such an optimization has been well established [1, 90, 91, 89, 52]. Some approaches to this optimization attempt to allocate objects on stack when possible [73, 81, 15, 16, 23]. This ensures that the memory is automatically deallocated when activation records are popped off the control stack.

Among earliest data flow analyses, Kennedy [55] presented liveness analysis for scalar variables and since then it has been discussed thoroughly in the literature. Liveness of heap data was first approximated by Agesen, Detlefs and Moss [1] by performing liveness of root variables on the stack that point to heap data. A more precise liveness analysis for heap cells was formulated recently by Khedker, Sanyal and Karkare [62].

The concept of aliasing was first studied in the context of interprocedural analysis for discovering the side effects of function calls. Cooper [25] introduced aliasing in the context formal parameters. Later aliasing of pointers was studied in details. We list references in the bibliographic notes of Chapter 4.

Cocke [24], Ullman [100], Allen [4, 5], and Kennedy [55, 56] were the earliest researchers in intraprocedural data flow analysis. The most influential work in intraprocedural analysis is the classical work by Kildall [63] and Kam and Ullman [49].

Spillman [94], Allen [6], Barth [13] and Banning [12] were the earliest researchers to study interprocedural data flow analysis. This was motivated by the side effect analysis. The most influential work on interprocedural data flow analysis is the classical work by Sharir and Pnueli [93].