Interprocedural Shape Analysis

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of
Master of Science in Computer Science

Noam Rinetzky

Submitted to the Senate of
the Technion – Israel Institute of Technology

Kislev 5761 Haifa December 2000

Interprocedural Shape Analysis

Noam Rinetzky

The research was done under the supervision of Dr. Shmuel Sagiv and Prof. Orna Grumberg in the department of Computer Science.

I would like to thank all those who helped me along the way.

I especially want to thank:

Dr. Mooly Sagiv for his guidance, support and endless optimism.

Prof. Orna Grumberg for her time and advice.

Prof. Thomas Reps and Prof. Reinhard Wilhelm for their useful comments.

Nurit Dor for the valuable discussions and suggestions.

Arie Freund, Tal Lev-Ami, Ran Shaham, Alex Warshavsky, and Eran Yahav for their reviews.

My family.

The generous financial help of the Technion and the Israeli Academy of Science is gratefully acknowledged.

Contents

Abstract **1**

1 Introduction **2**

1.1 Main Results . . . 3

1.1.1 A novel algorithm for interprocedural shape analysis of programs manipulating linked lists . . . 3

1.1.2 Scaling the algorithm . . . 7

1.1.3 Abstract data types . . . 7

1.2 Outline . . . 10

2 Calling Conventions **11**

3 The use of 3-valued logic for program analysis **13**

3.1 Representing memory states via 2-valued logical structures . . 13

3.2 Consistent 2-valued structures . . . 16

3.3 The meaning of programs statements . . . 17

3.4 Kleene's 3-valued logic . . . 17

3.5 Conservative representation of sets of memory states via 3-valued structures . . . 17

3.6 Expressing properties via formulae . . . 19

4 Analyzing Recursive Procedures **21**

4.1 Observing selected properties in order to improve the analysis 21

4.2 The best abstract transformer . . . 25

4.2.1 The abstraction principle . . . 25

4.2.2 Analyzing return statements . . . 26

4.3 Our iterative algorithm . . . 28

5 Towards a Scalable Shape Analysis Algorithm **30**
5.1 Overview 30
5.1.1 Outline 31
5.1.2 A motivating example 31
5.2 Limitations and simplifying assumptions 32
5.3 The algorithm 34
5.3.1 Call statements 34
5.3.2 Intraprocedural statements 35
5.3.3 Return statements 35
5.4 Discussion 37

6 Analysis of Programs Manipulating Abstract Data Types **39**
6.1 Overview 39
6.2 Analyzing a queue 39
6.2.1 A cheaper representation of the memory state 40
6.2.2 Representing the effect of a procedure by a finite state machine 44
6.2.3 The analysis 46
6.3 Experience with LEDA 46
6.4 Discussion 47

7 Prototype Implementation **49**
7.1 Empirical results 50

8 Conclusions and Future Work **52**
8.1 Current limitations and future work 54

Bibliography **55**

A Appendix **59**
A.1 Syntax of formulae 59
A.2 Kleene's 3-valued semantics 60
A.3 The meaning of programs statements 63
A.3.1 Intraprocedural statements 63
A.3.2 Interprocedural statements 66
A.4 Test cases. 69

List of Tables

3.1 The core predicates used in the analysis of linked list manipulating programns. 15
3.2 Kleene's 3-valued interpretation of the propositional operators. 17

4.1 The instrumentation predicates used in the interprocedural analysis of linked list manipulating programs. 22

6.1 The predicates used in the analysis of queue manipulating programs. 43

7.1 The total number of 3-valued structures that arise during analysis and running times for the recursive procedures analyzed. . 51

A.1 The predicate-update formulae defining the operational semantics of the intraprocedural statements. 65
A.2 The predicate-update formulae defining the operational semantics of the call and return statements. 67
A.3 3-valued formulae for conditions involving pointer variables. . 69

List of Figures

1.1 A type declaration for singly linked lists. 4
1.2 A recursive procedure which reverses a list. 4
1.3 The `main` procedure creates a list and then reverses it. 5
1.4 A program which creates two sets implemented as linked lists. 8
1.5 A procedure which inserts an element into a set implemented as a linked list 9

3.1 The 2-valued structure $S_{3.1}$ which corresponds to the program state at l_2 in the `rev` procedure upon exit of the fourth recursive invocation of the `rev` procedure. 16
3.2 The 3-valued structure $S_{3.2}$ which represents the 2-valued structure shown in Figure 3.1. 19

4.1 The 3-valued structure $S_{4.1}$ with instrumentation predicates which represents the 2-valued structure shown in Figure 3.1. . 24
4.2 The best abstract semantics of a statement `st` with respect to 3-valued structures. 25

5.1 Representative structures that occur in the analysis of procedure `set_insert` 33
5.2 An example for the *project* operator. 36
5.3 An example for the *combine* operator. 37

6.1 A declaration for the Queue type and the interface procedures that manipulate it. 41
6.2 An implementation of a Queue. 42
6.3 Representation of memory states by a *2-valued logical strucutre* as suggested at Chapter 4, compared to the representation suggested in this chapter. 43

6.4 A queue-manipulating program. 44
6.5 A non-deterministic FSM which represents the possible changes of the queue properties by invoking an interface procedure. . 45
6.6 The representation of the memory state at each point of the queue-manipulating program shown in Figure 6.4. 47

A.1 A recursive procedure which creates a list. 69
A.2 A non recursive procedure which adds one list to the end of another 70
A.3 A recursive procedure which frees all the elements of a list. . . 71
A.4 A recursive procedure which inserts an element into a sorted linked list. 72
A.5 A recursive procedure which deletes an element from a linked list. 73
A.6 A recursive procedure which searches for an element in a list. . 74
A.7 A recursive procedure which adds one list to the end of another. 75
A.8 A recursive procedure which reverses a list destructively. . . . 76

Abstract

A shape-analysis algorithm statically analyzes a program to determine information about the heap-allocated data structures that the program manipulates. The analysis algorithm is *conservative*, i.e., the discovered information is true for every input. The information can be used to understand, verify, optimize, or parallelize programs. For example, it can be utilized to check at compile-time for the absence of certain types of memory management errors, such as memory leakage or dereference of null pointers. Current shape analysis algorithms encounter severe difficulties when faced with programs composed of procedures, in particular recursive ones. Also, these algorithms are hard to scale.

We present a novel algorithm for interprocedural shape analysis of programs manipulating linked lists. Our algorithm analyzes programs invoking recursive procedures more precisely than any existing algorithm we know of. For example, it can verify the absence of memory leaks in many recursive programs, which is beyond the scope of existing algorithms. A prototype of the algorithm was implemented. It handles programs manipulating linked lists written in a subset of C.

Our algorithm handles arbitrary list-manipulating programs. However in practice, the analyzer might run out of space. We suggest a technique that improves the space (and time) requirements of our algorithm for non-recursive programs that manipulate several disjoint data structures. We look at this technique as a first step in an effort to scale shape analysis. In addition, we propose an even more efficient solution for programs that modify their heap allocated data structures only using a fixed set of interface procedures.

Chapter 1

Introduction

A shape-analysis algorithm statically analyzes a program to determine information about the heap-allocated data structures that the program manipulates. The analysis algorithm is *conservative*, i.e., the discovered information is true for every input. The information can be used to understand, verify, optimize [GH98], or parallelize [LH88, Hen90, AW93, PCK93, Zap99] programs. For example, it can be utilized to check at compile-time for the absence of certain types of memory management errors, such as memory leakage or dereference of null pointers [DRS98, DRS00].

In the past two decades, many "shape-analysis" algorithms have been developed [JM81, JM82, LH88, HPR89, CWZ90, Str92, AW93, PCK93, Wan94, SRW98]. The "quality", and thus the usability, of the information these algorithms can determine relies on the assumption that the number of variables is fixed. When this assumption is violated, as is the case when recursive procedures are used, the "quality" of the results, as well as the cost, of these algorithms deteriorates. This is a problem since recursion provides a natural way to manipulate linked data structures.

The analysis of large programs presents another problem. It is difficult to perform a "local" analysis at the procedure level since in general, any heap allocated memory cell can be *destructively updated* (modified) by any procedure. Thus, scaling the algorithm to analyze full programs and not just single procedures is difficult.

1.1 Main Results

We present a novel algorithm for interprocedural shape analysis of programs manipulating linked lists written in a subset of C. Our algorithm analyzes programs invoking recursive procedures more precisely than any existing algorithm we know of. For example, it can verify the absence of memory leaks in many recursive programs which is beyond the scope of existing algorithms. We also suggest a technique which can improve in practice the costs (space and time) of our algorithm for non-recursive programs that manipulate several data structures.

1.1.1 A novel algorithm for interprocedural shape analysis of programs manipulating linked lists

We present a novel interprocedural shape analysis algorithm for programs manipulating linked lists. Our algorithm analyzes recursive procedures more precisely than existing algorithms. For example, it is able to verify that all the recursive list-manipulating procedures of a small library we experimented with always return a list and never create memory leaks (see Chapter 7). In fact, not only can our algorithm verify that correct programs do not produce errors, it can also find interesting bugs in incorrect programs. For instance, it finds that the recursive procedure `rev` shown in Figure 1.2, which reverses a list (declared in Figure 1.1) returns an acyclic linked list and does not create memory leaks. Furthermore, if an error is introduced by removing the statement `x->n = NULL`, the resultant program creates a cyclic list, which leads to an infinite loop on some inputs. Interestingly, our analysis indicates this error. Such a precise analysis of the procedure `rev` is quite a difficult task since (i) `rev` is recursive, and thus there is no bound on the number of activation records that can be created when it executes; (ii) the global store is updated destructively in each invocation; and (iii) the procedure is not tail recursive: It sets the value of the local variable `x` before the recursive call and uses it as an argument to `app` after the call ends. No other shape-analysis algorithm we know of is capable of producing results with such a high level of precision for programs that invoke this, or similar, procedures.

A shape-analysis algorithm, like any other static program-analysis algorithm, is forced to represent execution states of potentially unbounded size in a bounded way. This process, often called *summarization*, naturally entails a loss of information. In the case of interprocedural analyses, it is also

```
/* list.h */
typedef struct node {
  int d;
  struct node *n;
} *L;
```

Figure 1.1: A type declaration for singly linked lists.

```
/* rev.c */
#include "list.h"
L rev(L x)
{
l0:
    L xn, t;
    if (x == NULL) return NULL;
    xn = x->n;
    x->n = NULL;
l1: t = rev(xn);
    return app(t, x);
l2:
}
```

Figure 1.2: A recursive procedure which reverses a list in two stages: reverse the tail of the original list and store the result in `t`; then append the original first element at the end of the list pointed to by `t`. The code for procedure `app` is given in Section A.4. We also analyzed this procedure with a recursive version of append (see Chapter 7.)

```
/* main.c */
#include "list.h"
void main()
{
  L hd, z ;
  hd = create(8);
l3:  z = rev(hd);
}
```

Figure 1.3: The `main` procedure creates a list and then reverses it. The code for procedure `create` is given in Section A.4.

necessary to summarize all incarnations of recursive procedures in a bounded way.

Shape-analysis algorithms can analyze linked lists in a fairly precise way, e.g., see [SRW99]. For an interprocedural analysis, we therefore follow the approach suggested in [JM82, Deu90] by summarizing activation records in essentially the same way linked list elements are summarized. By itself, this technique does not retain the precision we would like. The problem is with the (abstract) values obtained for local variables after a call. The abstract execution of a procedure call forces the analysis to summarize, and the execution of the corresponding return has the problem of recovering the information lost at the call. Due to the lack of enough information about the potential values of the local variables, the analysis must make overly conservative assumptions. For example, in the `rev` procedure, if the analysis is not aware of the fact that each list element is pointed to by no more than one instance of the variable `x`, it may fail to verify that `rev` returns an acyclic list (see Example 4.2.2).

An important concept in our algorithm is the identification of certain global properties of the heap elements pointed to by a local (stack-allocated) pointer variable. These properties describe potential and definite aliases between pointer access paths. This allows the analysis to handle return statements rather precisely. For example, in the `rev` procedure shown in Figure 1.2, the analysis determines that the list element pointed to by `x`

is different from all the list elements reachable from `t` just before the `app` procedure is invoked, which can be used to conclude that `app` must return an acyclic linked list. Proving that no memory leaks occur is achieved by determining that if an element of the list being reversed is not reachable from `t` at l_1, then it is pointed to by at least one instance of `x`.

A question that comes to mind is how our analysis determines such global properties in the absence of a specification. Fortunately, we found that a small set of "local" properties of the stack variables in the analyzed program can be used to determine many global properties. Furthermore, our analysis does not assume that a local property holds for the analyzed program. Instead, the analysis determines the stack variables that have a given property. Of course, it can benefit from the presence of a specification, e.g., [HHN92], which would allow us to look for the special global properties of the specified program.

For example, the property $sh_{\widehat{x}}(v)$ holds for a list element v that is pointed to by two or more invisible instances (see Chapter 2) of the parameter `x` from previous activation records. When $sh_{\widehat{x}}(v)$ does not hold for any list element, we have a guarantee that no list element is pointed to by more than one instance of the variable `x`. This simple local property plays a vital rule in verifying that the procedure `rev` returns an acyclic list (see Example 4.2.2). Interestingly, this property also sheds some light on the importance of tracking the sharing properties of stack variables. Existing intraprocedural shape-analysis algorithms [JM81, CWZ90, SRW98, SRW99] only record sharing properties of the heap since the number of variables is fixed in the intraprocedural setting. However, in the presence of recursive calls, different incarnations of a local variable may point to the same heap cell.

The ability to have distinctions between invisible instances of variables based on their local properties is the reason for the difference in precision between our method and the methods described in [LH88, Hen90, LH88, CWZ90, AW93, GH96, SRW98]. In Chapter 4, we also exploit properties that capture relationships between the stack and the heap. In many cases, the ability to have these distinctions also leads to a more efficient analysis.

Our algorithm was developed within the parametric framework described in [SRW99, LAS00]. That framework allows the generation of *intra*procedural shape-analysis algorithms based on an appropriate specification. In our work we show that their framework can generate quite precise *inter*procedural algorithms too. This solves an open problem mentioned there. Interestingly

our proposed solution is more precise and more efficient than the original interprocedural shape analysis algorithm [SRW98].

1.1.2 Scaling the algorithm

Our algorithm can handle any list-manipulating program. However, In practice the analyzer might run out of space. We address this problem and suggest a technique that can reduce the costs (space and time) of the analysis, for non-recursive programs that manipulates several disjoint data structures. For example, the program whose `main` procedure is shown in Figure 1.4 constructs two sets of integers. The sets are implemented as linked lists. Odd numbers are inserted into the list pointed to by `o` and even numbers are inserted into the list pointed to by `e`. The program manipulates the list by invoking the procedure `set_insert` shown in Figure 1.5. `set_insert` traverses the list pointed to by `x` searching for a list element with an integer value equal to `v`. If such an element is not found, a new list element is allocated and prepended to the list `x` points to. The procedure sets `rt` to point to the head (first element) of the list.

Although the program uses two sets, we observe that when `set_insert` is invoked, it manipulates only one of them. Since the lists are always disjoint, in any particular invocation of `set_insert` the lists not manipulated are "irrelevant"; they cannot affect the execution of the procedure and in particular cannot be modified.

We utilize this sort of observations, and analyze the behaviour of `set_insert` on a single linked list and adapt the result for the invocation in main, by inferring that the other list has not changed.

1.1.3 Abstract data types

We have also investigated a more efficient solution for programs which manipulate the heap in a "more controled way". The idea is that the application program uses abstract data types, ADTs, that are only modified using a fixed set of interface procedures. In particular, the application program does not directly access the content of the heap. Because of time constraints we only briefly studied this direction.

```
/* main.c */
#include "list.h"

L x = NULL, rt = NULL;
L g = NULL, o = NULL, e = NULL;
int v;


void main()
{
    ...
    /* g points to the head of a list */
    ...
    while (scanf("%d",&v) == 1)
    {
        if (v % 2 == 0) {
            x = e;
            l1 : set_insert(); l1^r :
            e = rt;
            rt = NULL;
        }
        else {
            x = o;
            l2 : set_insert(); l2^r :
            o = rt;
            rt = NULL;
        }
    }
}
```

Figure 1.4: A program which crates two sets o even, and odd integers. The sets are implemented as linked lists.

```
/* set_insert.c */
#include "list.h"

L t;

void set_insert()
{
    l^n : t = x;
    while (t!=NULL) {
        if (t->d == v) {
            rt = x ;
            t = NULL;
            return;
        }
        x = x->n;
    }
    t = malloc (sizeof(*L));
    t->d = v;
    t->n = x;
    rt = t;
    t = NULL;
    return;
l^x :
}
```

Figure 1.5: A procedure which inserts v into a set implemented as a linked list. The procedure prepends new elements at the beginning of the list.

1.2 Outline

Our algorithm has been developed using the 3-valued logic framework presented in [SRW99, LAS00]. This framework provides a sound theoretical foundation for our ideas and immediately leads to the prototype implementation Therefore, chapter 3 recalls some basic introduction to 3-valued logic. Chapter 4 presents our algorithm, in chapter 7 we describe its prototype implementation. Chapter 5 describes our technique to reduce the space our algorithm requires and Chapter 6 describes our solution for programs manipulating ADTs. In Chapter 8 we conclude and review related works. In Chapter 2 we define our calling conventions.

Chapter 2

Calling Conventions

In this chapter, we define our assumption about the programming language calling conventions. These conventions are somewhat arbitrary; in principle, different ones could be used with little effect on the capabilities of the program analyzer. Our analysis is not effected by the value of non-pointer variables. Thus, we do not represent scalars (conservatively assuming that any value is possible, if necessary), and in the sequel, restrict our attention to pointer variables.

Without loss of generality, we assume that all variables have unique names. Every invoked procedure has an activation record in which its local variables and parameters are stored. An invocation of procedure f at a *call-site label* is performed in several steps: (i) store the values of actual parameters and *label* in some designated global variables; (ii) at the *entry-point* of f, create a new activation record at the top of the stack and copy values of parameters and *label* into that record; (iii) execute the statements in f until a `return` statement occurs or f's *exit-point* is reached (we assume that a return statement stores the return value in a designated global variable and transfers the control to f's exit-point); (iv) at f's exit-point, pop the stack and transfer control back to the matching *return-site* of *label*; (v) at the return-site, copy the return value if needed and resume execution in the caller.

The activation record at the top of the stack is referred to as the *current activation record*. Local variables and parameters stored in the current activation record and global variables are called *visible*; local variables and parameters stored in other activation records are *invisible*.

Example 2.0.1 The C program whose `main` procedure shown in Figure 1.3 invokes `rev` shown in Figure 1.2 on a list with eight elements. In procedure `rev`, label l_1 plays the role of the recursive call site, l_0 that of `rev`'s entry point, and l_2 of `rev`'s exit point. This program is used throughout most of the thesis as a running example.

Chapter 3

The use of 3-valued logic for program analysis

The algorithm is explained (and implemented) using the 3-valued logic framework developed in [LAS00, SRW99]. In this section, we summarize that framework, which shows how 3-valued logic can serve as the basis for program analysis.

3.1 Representing memory states via 2-valued logical structures

A 2-*valued logical structure* S is comprised of a set of individuals (nodes) called a universe, denoted by U^S, and an interpretation over that universe for a set of predicate symbols called the *core predicates*. The interpretation of a predicate symbol p in S is denoted by p^S. For every predicate p of arity k, p^S is a function $p^S\colon (U^S)^k \to \{0, 1\}$.

In this thesis, 2-valued logical structures represent memory states. An individual corresponds to a memory element: either a heap cell (a list element) or an activation record. The core predicates describe atomic properties of the program memory state. The properties of each memory element are described by unary core predicates. The relations that hold between two memory elements are described by binary core predicates. The core predicates' intended meaning is given in Table 3.1. This representation intentionally ignores the specific values of pointer variables (i.e., the specific memory addresses that they contain), and record only certain relationships that hold among the

variables and memory elements:

- Every individual v represents either a heap cell in which case $heap^S(v) = 1$, or an activation record, in which case $stack^S(v) = 1$.
- The unary predicate cs_{label} indicates the call-site at which a procedure is invoked. Its similarities with the call-strings of [SP81] are discussed in Chapter 8.
- The unary predicate *top* is true for the current activation record.
- The binary relation n captures the `n`-successor relation between list elements.
- The binary relation *pr* connects an activation record to the activation record of the caller.
- For a local variable or parameter named `x`, the binary relation x captures its value in a specific activation record.

2-valued logical structures are depicted as directed graphs. A directed edge between nodes u_1 and u_2 that is labeled with binary predicate symbol p indicates that $p^S(u_1, u_2) = 1$. Also, for a unary predicate symbol p, we draw p inside a node u when $p^S(u) = 1$; conversely, when $p^S(u) = 0$ we do not draw p in u. For clarity, we treat the unary predicates *heap* and *stack* in a special way; we draw nodes u having $heap^S(u) = 1$ as circles to indicate heap elements; and we draw nodes having $stack^S(u) = 1$ as rectangles to indicate stack elements.[1]

Example 3.1.1 The 2-valued structure $S_{3.1}$ shown in Figure 3.1 corresponds to the memory state at program point l_2 in the `rev` procedure upon exit from the fourth invocation of the `rev` function in the running example. The five rectangular nodes correspond to the activation records of the five procedure invocations. Note that our convention is that a stack grows downwards. The current activation record (of `rev`) is drawn at the bottom with *top* written inside. The three activation records (of `rev`) drawn above it correspond to pending invocations of `rev`. The activation record of the `main` procedure is labeled with cs_{exit}, indicating it is the first procedure invoked.

[1] This can be formalized alternatively using many sorted logics. We avoided that for the sake of simplicity, and for similarity with [SRW99].

Predicate	Intended Meaning
$heap(v)$	v is a heap element
$stack(v)$	v is an activation record
$cs_{label}(v)$	*label* is the call-site of the procedure whose activation record is v
$g(v)$	The heap element v is pointed to by a global variable `g`
$n(v_1, v_2)$	The `n`-component of list element v_1 points to the list element v_2
$top(v)$	v is the current activation record
$pr(v_1, v_2)$	The activation record v_2 is the immediate previous activation record of v_1 in the stack
$x(v_1, v_2)$	The local (parameter) variable `x`, which is stored in activation record v_1, points to the list element v_2

Table 3.1: The core predicates used in the analysis of linked list manipulating programs. There is a separate predicate g for every global program variable `g`, x for every local variable or parameter `x`, and cs_{label} for every label *label* immediately preceding a procedure call.

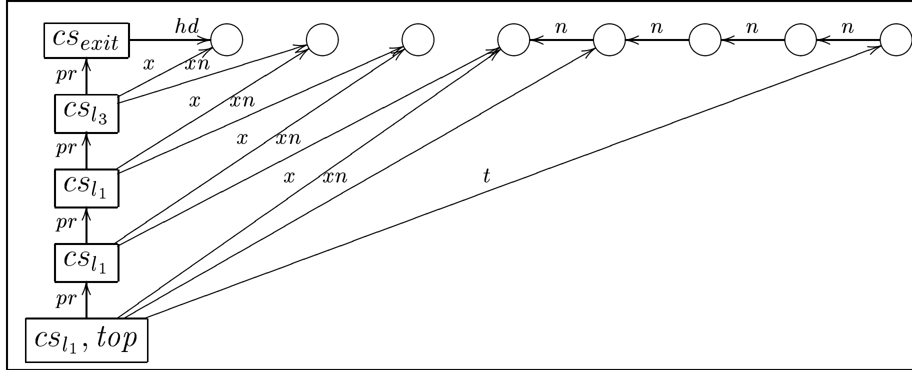

Figure 3.1: The 2-valued structure $S_{3.1}$ which corresponds to the program state at l_2 in the `rev` procedure upon exit of the fourth recursive invocation of the `rev` procedure.

The three isolated heap nodes on the left side of the figure correspond to the list elements pointed to by `x` in pending invocations of `rev`. The chain of five heap nodes to the right correspond to the (reversed) part of the original list. The last element in the list corresponds to the list element appended by `app` invoked just before l_2 in the current invocation of `rev`.

Notice that the `n` predicate is the only one that is specific to the linked list structure declared in Figure 1.1. The remaining predicates would play a rule in the analysis of any data structure.

3.2 Consistent 2-valued structures

Some 2-valued structures cannot represent memory states, e.g., when a unary predicate g holds at two different nodes for a global variable `g`. A 2-valued structure is *consistent* if it can represent a memory state. It turns out that the analysis can be more precise by eliminating inconsistent 2-valued structures. Therefore, in Section 4.3 we sketch a constructive method to check if a 2-valued structure is inconsistent and thus can be discarded by the analysis.

$\wedge$	0	1	1/2
0	0	0	0
1	0	1	1/2
1/2	0	1/2	1/2

$\vee$	0	1	1/2
0	0	1	1/2
1	1	1	1
1/2	1/2	1	1/2

$\neg$	
0	1
1	0
1/2	1/2

Table 3.2: Kleene's 3-valued interpretation of the propositional operators.

3.3 The meaning of programs statements

The meaning functions for program statements are defined as transformers from 2-valued structures to 2-valued structures. Properties of memory states can be obtained by evaluating first order logical formula against the representing structure (see Section 3.6), thus, these transformers are defined by collection of first order formulae evaluated against the original structure. The value of every predicate is determined by a corresponding formula. The main idea is that if a structure S represents a set of memory states that arise before statement st, than a structure S' that represents the corresponding set of stores that arise after st can be obtained by evaluating a suitable collection of formulae that capture the semantics of st. Section A.3 defines the operational semantics of the subset of the C programming language analyzed.

3.4 Kleene's 3-valued logic

Kleene's 3-valued logic is an extension of ordinary 2-valued logic with the special value of 1/2 (unknown) for cases in which predicates could have either value, i.e., 1 (true) or 0 (false). Kleene's interpretation of the propositional operators is given in Table 3.2. We say that the values 0 and 1 are *definite values* and that 1/2 is an *indefinite value*.

3.5 Conservative representation of sets of memory states via 3-valued structures

Like 2-valued structures, a 3-*valued logical structure* S is also comprised of a universe U^S and an interpretation of the predicate symbols. However, for

every predicate p of arity k, p^S is a function $p^S\colon (U^S)^k \rightarrow \{0, 1, 1/2\}$, where 1/2 explicitly captures unknown predicate values.

3-valued logical structures are also drawn as directed graphs. Definite values are drawn as in 2-valued structures. Binary indefinite (1/2) predicate values are drawn as dotted directed edges. Also, we draw $p = 1/2$ inside a node u when $p^S(u) = 1/2$.

Let $S^\natural$ be a 2-valued structure, S be a 3-valued structure, and $f\colon U^{S^\natural} \rightarrow U^S$ be a surjective function. We say that f *embeds* $S^\natural$ *into* S if for every predicate p of arity k and $u_1, u_2, \ldots, u_k \in U^{S^\natural}$, either $p^{S^\natural}(u_1, u_2, \ldots, u_k) = p^S(f(u_1), f(u_2), \ldots, f(u_k))$ or $p^S(f(u_1), f(u_2), \ldots, f(u_k)) = 1/2$. We say that S *conservatively represents all the* 2*-valued structures that can be embedded into it by some function* f. Thus, S can compactly represent many structures.

Nodes in a 3-valued structure that may represent more than one individual from a given 2-valued structure are called *summary nodes.* We use a designated unary predicate sm to maintain summary-node information. A summary node w has $sm^S(w) = 1/2$, indicating that it may represent more than one node from 2-valued structures. These nodes are depicted graphically as dotted circles or rectangles. In contrast, if $sm^S(w) = 0$, then w is known to represent a unique node. We impose an additional restriction on embedding functions: only nodes with $sm^S(w) = 1/2$ can have more than one node mapped to them by an embedding function.

Example 3.5.1 The 3-valued structure $S_{3.2}$ shown in Figure 3.2 represents the 2-valued structure $S_{3.1}$ shown in Figure 3.1. The dotted circle summary node represents all the eight list elements. The indefiniteness of the self n-edge results from the fact that there is an n-component pointer between each two successors and no n-component pointer between non-successors.

The dotted rectangle summary node represents the activation records from the second and third invocation of `rev`. The unary predicate cs_{l_1} drawn inside it indicates that it (only) represents activation records of `rev` that invoked at l_1 (i.e., recursive calls). The dotted x-edge from this summary node indicates that an invisible instance of `x` from the second or the third call may or may not point to one of the list elements. The rectangle at the top of Figure 3.2 represents the activation record at the top of Figure 3.1, which is the invocation of `main`. The second rectangle from the top in $S_{3.2}$ represents the second rectangle from the top in $S_{3.1}$ which is an invocation of `rev` from `main` (indicated by the occurrence of cs_{l_3} inside this node). The bottom rectangle in $S_{3.2}$ represents the bottom rectangle in $S_{3.1}$, which is

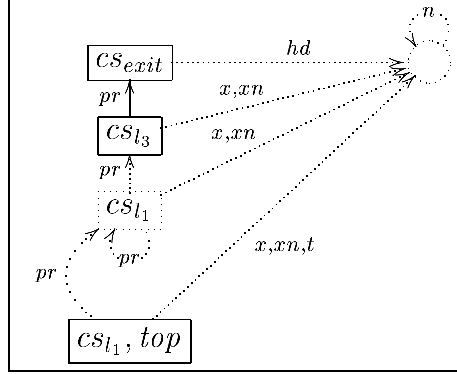

Figure 3.2: The 3-valued structure $S_{3.2}$ which represents the 2-valued structure shown in Figure 3.1.

the current activation record (indicated by the occurrence of *top* inside this node. All other activation records are known not to be the current activation record (i.e., the *top* predicate does not hold for these nodes) since *top* does not occur in either of them.

3.6 Expressing properties via formulae

Properties of structures can be extracted by evaluating formulae. We use first-order logic with transitive closure and equality, but without function symbols and constant symbols.[2] The formal definition for the syntax of formulae is in A.1 and the definition for kleene's 3-valued semantics is in A.2.

For example, the formula

$$\begin{aligned} \exists v_1, v_2 : \quad & \neg top(v_1) \wedge \neg top(v_2) \wedge v_1 \neq v_2 \wedge \\ & x(v_1, v) \wedge x(v_2, v) \end{aligned} \tag{3.1}$$

expresses the fact that there are two different invisible instances of the parameter variable `x` pointing to the same list element v.

The Embedding Theorem (see [SRW99, Theorem 3.7]) states that any formula that evaluates to a definite value in a 3-valued structure evaluates to the same value in all of the 2-valued structures embedded into that structure.

[2] There is one non-standard aspect in our logic; $v_1 = v_2$ and $v_1 \neq v_2$ are indefinite in case v_1 and v_2 are the same summary node. The reason for this is seen shortly.

The Embedding Theorem is the foundation for the use of 3-valued logic in static-analysis: It ensures that it is sensible to take a formula that—when interpreted in 2-valued logic—defines a property, and reinterpret it on a 3-valued structure S: The Embedding Theorem ensures that one must obtain a value that is conservative with regard to the value of the formula any 2-valued structure represented by S.

Example 3.6.1 Consider the 2-valued structure $S_{3.1}$ shown in Figure 3.1. The formula (3.1) evaluates to 0 at all of the list nodes.

In contrast, consider the 3-valued structure $S_{3.2}$ shown in Figure 3.2. This formula (3.1) evaluates to 1/2 at the dotted circle summary heap node. This is in line with the Embedding Theorem since 1/2 is not a definite value. However, it is not very precise since the fact that different invisible instances of `x` are never aliased is lost.

Chapter 4

Analyzing Recursive Procedures

In this chapter, we describe our shape-analysis algorithm for recursive programs manipulating linked lists. The algorithm iteratively annotates each program point with a set of 3-valued logical structures in a *conservative* manner, i.e., when it terminates, every 2-valued structure that can arise at a program point is represented by one of the 3-valued structures computed at this point. However, it may also conservatively include superfluous 3-valued structures.

Section 4.1 describes the properties of heap elements and local variables which are tracked by the algorithm. For ease of understanding, in Section 4.2, we give a high-level description of the iterative analysis algorithm. The actual algorithm is presented in Section 4.3.

4.1 Observing selected properties in order to improve the analysis

To overcome the kind of imprecision described in Example 3.6.1, we introduce *instrumentation predicates*. These predicates are stored in each structure, just like the core predicates. The values of these predicates are derived from the core predicates, that is, every instrumentation predicate has a formula over the set of core predicates that defines its meaning. The instrumentation predicates that our interprocedural algorithm utilizes are described in Table 4.1, together with their informal meaning and their defining formula

Predicate	Intended Meaning	Defining Formula
$x(v)$	The list element v is pointed to by the visible instance of `x`.	$\exists v_1 : top(v_1) \wedge x(v_1, v)$
$r_{n,x}(v)$	The list element v is reachable by following `n`-components from the visible instance of `x`.	$\exists v_1, v_2 : top(v_1) \wedge x(v_1, v_2) \wedge n^*(v_2, v)$
$\widehat{x}(v)$	The list element v is pointed to by an invisible instance of `x`.	$\exists v_1 : \neg top(v_1) \wedge x(v_1, v)$
$r_{n,\widehat{x}}(v)$	The list element v is reachable by following `n`-component from an invisible instance of `x`.	$\exists v_1, v_2 : \neg top(v_1) \wedge x(v_1, v_2) \wedge n^*(v_2, v)$
$sh_{\widehat{x}}(v)$	The list element v is pointed to by more than one invisible instance of `x`.	$\exists v_1, v_2 : v_1 \neq v_2 \wedge \neg top(v_1) \wedge x(v_1, v) \wedge \neg top(v_2) \wedge x(v_2, v)$
$nn_{\widehat{x}}(v)$	The invisible instance of `x` stored in the activation record v points to some list element.	$\exists v_1 : \neg top(v) \wedge x(v, v_1)$
$al_{x,y}(v)$	The invisible instances of `x` and `y` stored in the activation record v are aliased.	$\exists v_1 : \neg top(v) \wedge x(v, v_1) \wedge y(v, v_1)$
$al_{x,pr[y]}(v)$	The instance of `x` stored in the activation record v is aliased with the instance of `y` stored in v's previous activation record.	$\exists v_1, v_2 : pr(v, v_1) \wedge x(v, v_2) \wedge y(v_1, v_2)$
$al_{x,pr[y]\rightarrow n}(v)$	The instance of `x` stored in the activation record v is aliased with `y->n` for the instance of `y` stored in v's previous activation record	$\exists v_1, v_2, v_3 : pr(v, v_2) \wedge y(v_2, v_3) \wedge n(v_3, v_1) \wedge x(v, v_1)$
$al_{x\rightarrow n,pr[y]}(v)$	`x->n` for the instance of `x` stored in the activation record v is ali- -ased with the instance of `y` stored in v's previous activation record.	$\exists v_1, v_2, v_3 : pr(v, v_3) \wedge y(v_3, v_1) \wedge x(v, v_2) \wedge n(v_2, v_1)$

Table 4.1: The instrumentation predicates used for the interprocedural analysis. Here x and y are generic names for local variables and parameters `x` and `y` of an analyzed function. The n^* notation used in the defining formula for $r_{n,x}(v)$ denotes the reflexive transitive closure of n.

(other intraprocedural instrumentation predicates are defined in [SRW99]).

The instrumentation predicates are divided into four classes, separated by double horizontal lines in Table 4.1: (i) Properties of heap elements with respect to visible variables, i.e., x and $r_{n,x}$. These are the ones originally used in [SRW99]. (ii) Properties of heap elements with respect to invisible variables. These are $\widehat{x}$ and $r_{n,\widehat{x}}$, which are variants of x and $r_{n,x}$ from the first class, but involve the invisible variables. The $sh_{\widehat{x}}(v)$ predicate is motivated by Example 3.6.1. It is similar to the heap-sharing predicate used in [JM81, CWZ90, SRW98, SRW99]. (iii) Generic properties of an individual activation record. For example, $nn^S_{\widehat{x}}(u) = 1$ (nn for not NULL) in a 2-valued structure S indicates that the invisible instance of `x` that is stored in activation record u points to some list element. (iv) Properties across successive recursive calls. For example, the predicate $al_{x,pr[y]}$ captures aliasing between `x` at the callee and `y` at the caller. The other properties are similar but also involve the `n` component.

Example 4.1.1 The 3-valued structure $S_{4.1}$ shown in Figure 4.1 also represents the 2-valued structure $S_{3.1}$ shown in Figure 3.1. In contrast with $S_{3.2}$ shown in Figure 3.2, in which all eight list elements are represented by one heap node, in $S_{4.1}$, they are represented by six heap nodes. The leftmost heap node in $S_{4.1}$ represents the leftmost list element in $S_{3.1}$ (which was originally the first list element). The fact that $\widehat{hd}$ is drawn inside this node indicates that it represents a list element pointed to by an invisible instance of `hd`. This fact can also be extracted from $S_{4.1}$ by evaluating the $\widehat{hd}$ defining formula at this node, but this is not always the case, as we will now see: The second leftmost heap node is a summary node that represents both the second and third list elements from the left in $S_{3.1}$. There is an indefinite x-edge into this summary node. Still, since $\widehat{x}$ is drawn inside it, every list element it represents must be pointed to by at least one invisible instance of `x`. Therefore, the analysis can determine that this node does not represent storage locations that have been leaked by the program.

The other summary heap node (the second heap node from the right) represents the second and third (from the right) list elements of $S_{3.1}$. Its incoming n edge is indefinite. Still, since $r_{n,t}$ occurs inside this node, we know that all the list elements it represents are reachable from `t`.

Note that the predicate $sh_{\widehat{x}}$ does not hold for any heap node in $S_{4.1}$. Therefore, no list element in any 2-valued structure that $S_{4.1}$ represents is pointed to by more than one invisible instance of the variable `x`. Note that

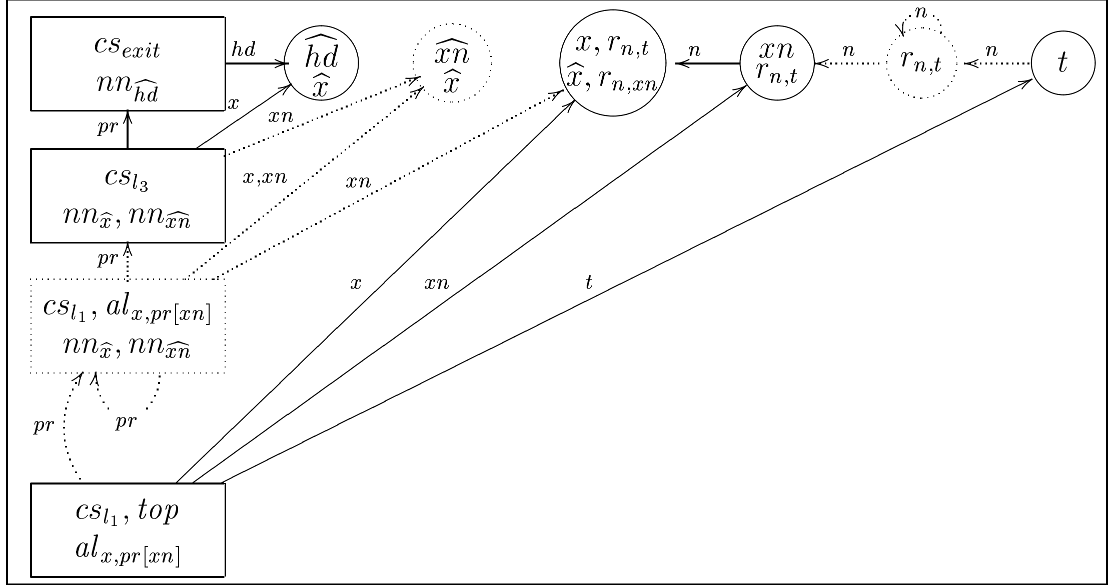

Figure 4.1: The 3-valued structure $S_{4.1}$ with instrumentation predicates which represents the 2-valued structure shown in Figure 3.1. For brevity, we do not show $r_{n,x}$ for nodes having the property x.

the combination of $sh_{\widehat{x}}(u) = 0$ (pointed to by ≤ 1 invisible instance of `x`) and $\widehat{x}() = 1$ (pointed to by ≥ 1 invisible instance of `x`) allows determining that each node represented by a summary node u is pointed to by exactly one invisible instance of `x` (cf. the second leftmost heap node in $S_{4.1}$).

The stack elements are depicted in the same way as they are depict by $S_{3.2}$ (see Example 3.5.1). Since $al_{x,pr[xn]}$ occurs inside the two stack nodes at the bottom, for every activation record v they represent, the instance of `x` stored in v is aliased with the instance of `xn` stored in the activation record preceding v.

Notice that the $r_{n,x}$, $r_{n,\widehat{x}}$, $al_{x,pr[y]\rightarrow n}$, $al_{x\rightarrow n,pr[y]}$ predicates are the only instrumentation predicates specific to the linked list structure declared in Figure 1.1. The remaining predicates would play a role in any analysis that would attempt to analyze the runtime stack.

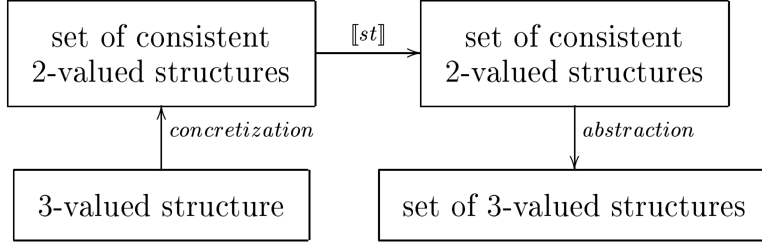

Figure 4.2: The best abstract semantics of a statement `st` with respect to 3-valued structures. $[\![st]\!]$ is the operational semantics of `st` applied pointwise to every consistent 2-valued structure.

4.2 The best abstract transformer

This section provides a high level description of the algorithm in terms of the general abstract interpretation framework [CC79]. Conceptually, the most precise (also called *best*) conservative effect of a program statement on a 3-valued logical structure S is defined in three stages shown in Figure 4.2: (i) find each consistent 2-valued structure $S^\natural$ represented by S (*concretization*); (ii) apply the C operational semantics to every such structure $S^\natural$ resulting in a 2-valued structure $S^{\natural\prime}$ and (iii) finally abstract each of the 2-valued structures $S^{\natural\prime}$ by a 3-valued structure of bounded size (*abstraction*). Thus, the result of the statement is a set of 3-valued structures of bounded size.

4.2.1 The abstraction principle

The abstraction function is defined by a subset of the unary predicates, that are called *abstraction properties* in [SRW99]. The abstraction of a 2-valued structure is defined by mapping all the nodes which have the same values for the abstraction properties into the same abstract node. Thus, the values of abstraction predicates remain the same in the abstracted 3-valued structures. The values of every other predicate p in the abstracted 3-valued structure are determined conservatively to yield an indefinite value whenever corresponding values of p in the represented concrete 2-valued structure disagree.

Example 4.2.1 The structure $S_{3.2}$ shown in Figure 3.2 is an abstraction of $S_{3.1}$ shown in Figure 3.1 when all of the unary core predicates are used as abstraction properties. For example, the activation records of the 2nd and

3rd recursive call to `rev` are both mapped into the summary stack node since they are both invisible activation records of invocations of `rev` from the same call-site (i.e., *top* does not hold for these activation records, but cs_{l_1} does). Also, all of the eight heap nodes are mapped to the same summary-node for which only the *heap* core predicate holds. The *pr*-edge into the stack node at the top of the figure is definite since there is only one node mapped to each of the edge's endpoints. In contrast, the *hd*-edge emanating from the uppermost stack node must be indefinite in order for $S_{3.2}$ to conservatively represent $S_{3.1}$: In $S_{3.1}$ the *hd* predicate holds for the uppermost stack node and the leftmost heap node, but it does not hold for any other heap node, and all heap nodes of $S_{3.1}$ are summarized into one summary heap node.

The structure $S_{4.1}$ shown in Figure 4.1 is an abstraction of $S_{3.1}$ shown in Figure 3.1 when the abstraction properties all of the unary core and instrumentation predicates. Notice that nodes with different observed properties lead to different instrumentation predicate values and thus are never represented by the same abstract node.

Because the set of unary predicates is fixed, there can only be a constant number of nodes in an abstracted structure which guarantees that the analysis always terminates.

4.2.2 Analyzing return statements

How to retain precision when the analysis performs its abstract execution across a return statement is the key problem that we face. By exploiting the instrumentation predicates, our technique is capable of handling return statements quite precisely. This is demonstrated in the following example. For expository purposes we will explain the abstract execution of return statement in term of the best abstract transformer, described in Section 4.2. The actual method our analysis uses is discussed in Section 4.3.

Example 4.2.2 Let us exemplify the application of the return statement to the 3-valued structure $S_{4.1}$ shown in Figure 4.1 following the stages of the *best* iterative algorithm described above.

Stage I–Concretization

Let $S^\natural$ be one of the consistent 2-valued structures represented by $S_{4.1}$. Let $k \geq 1$ be the number of activation records represented by the summary stack

node in $S_{4.1}$. Since $S^\natural$ is a consistent 2-valued structure, the x parameter variable in each of these k activation records must point to one of the isolated list elements represented by the left summary heap node. This can be inferred by the following series of observations: the fact that the x variable in each of these activation records points to a list element is indicated by the presence of $nn_{\widehat{x}}$ inside the stack summary node. The list elements pointed to by these variables must be represented by the left summary heap node since only one x-edge emanates from the summary stack node, and this edge enters the left summary heap node.

Let $m \geq 1$ be the number of list elements represented by the left summary heap node. Since $\widehat{x}$ occurs inside this node, each of the m list elements it represents must be pointed to by at least one invisible instance of x. Thus, $m \leq k$. However since $sh_{\widehat{x}}$ does not occur inside this summary node, none of the m list elements it represents is pointed to by more than one invisible instance of x. Thus, we conclude that $m = k$.

Using the fact that $al_{x,pr[xn]}$ is drawn inside the two stack nodes at the bottom of Figure 4.1, we conclude that the instance of x in each recursive invocation of rev is aliased with the instance of xn of rev's previous invocation. Thus, each acceptable $S^\natural$ looks essentially like the structure shown in Figure 3.1, but with k isolated list elements not pointed to by hd, rather than two, and with some number of elements in the list pointed to by x.

Stage II–Applying the operational semantics

Applying the operational semantics of return to $S^\natural$ (see Chapter 2 and Section A.3) results in a (consistent) 2-valued structure $S^{\natural\prime}$. Note that the list element pointed to by the visible instance of x in $S^{\natural\prime}$ is not pointed to by any other instance of x, and it is not part of the reversed suffix. Thus $S^{\natural\prime}$ differs from $S^\natural$ by having the top activation record of $S^\natural$ removed from the stack and by having the activation record preceding it be the new *current* activation record.

Stage III–abstraction

Abstracting $S^{\natural\prime}$ into a 3-valued structure may result, depending on k, in one of three possible structures. If $k > 2$ then the resulting structure is very similar to $S_{4.1}$, since the information regarding the number of remaining isolated list elements and invisible activation record is lost in the summarization. For

$k = 1$ and $k = 2$ we have a consistent 2-valued structures with four and three activation records, respectively. Abstracting these structures results in no summary stack nodes, since the call-site of each non-current activation record is different. For $k = 1$ only one isolated list elements remains, thus it is not summarized. For $k = 2$ the two remaining isolated heap nodes are not merged since they are pointed to by different (invisible) local variables. For example, one of them is pointed to by `hd` and the other one is not.

Notice that if no instrumentation predicates correlating invisible variables and heap nodes are maintained, a conservative analysis cannot deduce that the list element pointed to by the visible instance of `x` in $S^{\natural\prime}$ is not pointed to by another instance of this variable. Thus, the analysis must conservatively assume that future calls to `app` may create cycles. However, even when $al_{x,pr[xn]}$ is not maintained, the analysis can still produce fairly accurate results using only the $sh_{\widehat{x}}$ and $\widehat{x}$ instrumentation predicates.

4.3 Our iterative algorithm

Unlike a hypothetical algorithm based on the best abstract transformer which explicitly applies the operational semantics to each of the (potentially infinite) structures represented by a three-valued structure S, our algorithm explicitly operates on S, yielding a set of 3-valued structures S'. By employing a set of judgements, similar in spirit to the ones described Example 4.2.2 our algorithm produces a set which conservatively represents all the structures that could arise after applying the `return` statement to each consistent 2-valued structure S represents. However, in general, the transformers used are conservative approximations of the best abstract transformer; the set of structures obtained may represent more 2-valued structures than those represented by applying the best abstract transformer. Our experience to date, reported in Section 7, indicates that it usually gives good results.

Technically, the algorithm computes the resulting set of 3-valued structures S' by evaluating formulae in 3-valued logic. When interpreted in 2-valued logic these formulae define the operational semantics. Thus, the Embedding Theorem (see [SRW99, Theorem 3.7]) guarantees that the results are conservative w.r.t a hypothetical algorithm based on the best abstract transformer. The update formulae for the core-predicates describing the operational semantics is given in Section A.3.

We glossed over some several important details needed for boosting the

precision of our algorithm that can be found in Chapter 7.

Chapter 5

Towards a Scalable Shape Analysis Algorithm

5.1 Overview

In this chapter, we describe an attempt to improve the efficiency of the interprocedural shape analysis algorithm described in Chapter 4. The main idea is to decrease the space (and the time) of the algorithm by reducing the size of the structures used. Our chief insight is that parts of structures that represent pieces of information that cannot be affected by a procedure can be represented more conservatively during the analysis of the procedure body. The structures at a procedure return-site can be constructed by combining the structures at the exit-site with the unchanged parts of the structures at the call-site.

In order to find the "irrelevant" parts of the memory-state at the call-site, we treat the heap as an undirected graph; the nodes in this graph are the heap elements, and the graph edges are pointer components of heap elements. Any connected component that the procedure cannot refer to any of its nodes is *irrelevant* to the analysis of the procedure behaviour. Irrelevant parts can be conservatively found by analyzing the *3-valued logical strucutres* that arise in the analysis.

The new algorithm may be less precise than the one described in Chapter 4, although we do not expect this to happen. For details, see Section 5.4.

5.1.1 Outline

In Section 5.1.2 we give an informal description of the algorithm by means of an example. In Section 5.2 we restrict the set of programs that the algorithm can handle. Section 5.3 gives a formal description of the algorithm, and Section 5.4 concludes.

5.1.2 A motivating example

We use the program that creates two sets of integers described in Section 1.1.2 as a running example in this chapter. The algorithm presented in Chapter 4 analyzes `set_insert` with all the relevant structures, i.e., any combination of the three disjoint lists that arises at any call-site (notice that at l_1 and l_2, `g` points to a list). However, we observe that any particular invocation of `set_insert` may only refer to list elements reachable from the global variable `x`. Instead of analyzing `set_insert` in all combinations, we analyze the behaviour of `set_insert` on a single linked list and adapt the result for the invocation in main, by combining the unchanged linked lists.

Figure 5.1 compares the approach suggested in Chapter 4 with the one presented in this chapter. The second column indicates representative structures that occur when `set_insert` is invoked on a new set element. The third column shows the corresponding structures that arise in the analysis proposed in this chapter which uses a compact representation of "irrelevant" portions of the heap. When the same structures occur in both algorithms they are shown only once. To make the diagrams more intuitive, the value of the x-predicate which captures the "pointed-to by global variable `x`" property is depicted via an edge from a label x to the node `x` points to (and via the absence of edges from that label to the nodes `x` does not point to). Also, to avoid clutter, the activation record nodes are not shown.

When `set_insert` is entered at l^n, the algorithm described in the previous chapter, simply adds a new activation record, which is not shown. In particular, there is no change between the representation of the heap in S^1 and in S^2. In contrast, our new algorithm also compacts the representation of all the "irrelevant" calling context to one summary node. The new algorithm discovers that the four upper nodes in S^1 are irrelevant for the analysis of `set_insert`; they are not on any undirected path of edges from any node that is pointed-to by one of the variables that `set_insert` refers to: `x`, `t`, or `rt`. Thus, these nodes are represented by the same summary node in S^3.

On the other hand, the two lower nodes in S^1 that represent the head and the tail of the list pointed to by `x` are not compacted. The main idea is that properties of irrelevant nodes cannot be altered by any procedure invocation.

The new auxiliary property ic distinguishes nodes that represent "irrelevant context" from the rest of the nodes. In S^3, only the upper summary node has this property.

Intraprocedural statements are handled in the same way in both algorithms, resulting in S^4 and S^5 at the exit-site.

The structure S^4 arises at the exit-site of `set_insert`. The algorithm described in the previous chapter, creates the structure S^6 by popping the top activation record in S^4. Since the diagrams only show the heap , no difference is shown between S^4 and S^6. In contrast, our new algorithm constructs S^6 by combining S^1 and S^5. It takes the four upper nodes in S^1 and the three lower nodes in S^5 and produces S^6. In fact, the algorithm replaces the compact representation of the irrelevant context in S^5, by the more detailed one found in S^1. Notice that the structure S^6 is more precise than S^5 (ignoring the activation records that are not shown, and the auxiliary predicate ic).

The crucial differences between the two approaches are the fact that the new algorithm uses *both* S^1 and S^5 to produce S^6, and that the new algorithm, utilizes the fact that irrelevant information can be copied from S^1 into S^6.

5.2 Limitations and simplifying assumptions

The main limitation of our algorithm is that it cannot handle recursive procedures. For simplicity, in the rest of this chapter we assume that the analyzed programs adhere to the following restrictions: (a) procedures have no parameters, and (b) all variables are global. Even for restricted programs such as our running example, the algorithm presents space improvement over the algorithm presented in Chapter 4. When applied to our running example, the algorithm presented in this chapter uses only one node to represent the irrelevant context, while the algorithm of the previous chapter, uses two to four nodes for this task.

Loc	Without Compaction	With Compaction
l_2 :	(S^1) e $r_{n,e}$ n $r_{n,e}$ n g $r_{n,g}$ n $r_{n,g}$ n x, o $r_{n,x}$ $r_{n,o}$ n $r_{n,x}$ $r_{n,o}$ n	
l^n :	(S^2) e $r_{n,e}$ n $r_{n,e}$ n g $r_{n,g}$ n $r_{n,g}$ n x, o $r_{n,x}$ $r_{n,o}$ n $r_{n,x}$ $r_{n,o}$ n	(S^3) n g, e ic $r_{n,g}=1/2$ $r_{n,e}=1/2$ x, o $r_{n,x}$ $r_{n,o}$ n $r_{n,x}$ $r_{n,o}$ n
...	continue the same analysis algorithm	
l^x :	(S^4) e $r_{n,e}$ n $r_{n,e}$ n g $r_{n,g}$ n $r_{n,g}$ n rt $r_{n,rt}$ n $r_{n,x}$ $r_{n,rt}$ $r_{n,o}$ n $r_{n,x}$ $r_{n,rt}$ $r_{n,o}$ n x, o	(S^5) n g, e ic $r_{n,g}=1/2$ $r_{n,e}=1/2$ rt $r_{n,rt}$ n $r_{n,x}$ $r_{n,rt}$ $r_{n,o}$ n $r_{n,x}$ $r_{n,rt}$ $r_{n,o}$ n x, o
l_2^r	(S^6) e $r_{n,e}$ n $r_{n,e}$ n g $r_{n,g}$ n $r_{n,g}$ n rt $r_{n,rt}$ n $r_{n,x}$ $r_{n,rt}$ $r_{n,o}$ n $r_{n,x}$ $r_{n,rt}$ $r_{n,o}$ n x, o	

Figure 5.1: Representative structures that occur in the analysis of `set_insert` in the algorithm presented in Chapter 4 (without compaction) and the corresponding structures in the algorithm presented in this Chapter (with compaction.)

5.3 The algorithm

In this section, we describe our new algorithm. It is similar to the algorithm given in Chapter 4, but the analysis of call and return statements is changed.

5.3.1 Call statements

Given a structure S^c at a call-site of a procedure pr, the algorithm generates a structure S^n at the entry-site of procedure pr by adding a new activation record to S^c at the top of activation record stack (as described in Chapter 4), and *compacting* "irrelevant calling context". *Compacting* "irrelevant calling context" is done by (a) finding "irrelevant" nodes in U^{S^c} and (b) summarizing these nodes by one "irrelevant context" node.

A node u in U^{S^c} is *relevant for the analysis of procedure* pr if u represents a list element that appears on a (possibly empty) undirected path of n-selectors from some variable that pr refers to. The property "being relevant for procedure pr" is expressed by the formula:

$$r_{pr}(v) = \exists v_1 : uses_{Z_{pr}}(v_1) \wedge un^*(v_1, v)$$

The formula $uses_Z(v)$ is defined for every subset of the program variables $Z \subseteq PVar$ as $uses_Z(v) = \bigvee_{x \in Z} x(v)$; Z_{pr} denotes the subset of the global program variables that pr refers to. Thus, $uses_{Z_{pr}}(v)$ holds only at nodes that represent list elements pointed-to by a variable which is referred by pr. For example, since $Z_{set_insert} = \{x, t, rt\}$, the formula $uses_{Z_{set_insert}}(v)$ holds only at the leftmost node at the bottom in S^1 shown in Figure 5.1.

The formula $un^*(v_1, v_2)$ is a shorthand for reflexive transitive closure of the formula $un(v_1, v_2) = n(v_1, v_2) \vee n(v_2, v_1)$. The formula $un(v_1, v_2)$ expresses the property that v_1 is the n-successor of v_2 or vice-versa. The formula $un^*(v_1, v_2)$ captures the property that v_1 and v_2 are *connected* via an undirected path of n-selectors. For example, in S^1 the formula $un^*(v_1, v_2)$ evaluates to 1/2 for the two nodes at the bottom, indicating that they might be connected.

The algorithm chooses as the *irrelevant nodes* for procedure pr the nodes at which the formula $r_{pr}(v)$ evaluates to 0. For example, in S^1, shown in Figure 5.1, the formula r_{set_insert} evaluates to 0 at all the four upper nodes since there is no path of n-edges between the leftmost node at the bottom, which has the $uses_{Z_{set_insert}}$ property, and any of these nodes. Thus, these

nodes are irrelevant. Notice that even if r_{pr} evaluates to 1/2 at a node u, this node is not irrelevant and thus will be explicitly represented.

The algorithm summarizes the irrelevant nodes by one summary node by loosing information when necessary. This node has a definite value for a property p only if all the irrelevant nodes has that same value for p, and an indefinite value otherwise. For example, in Figure 5.1, S^3 results by compacting the irrelevant context of procedure `set_insert` in S^1. The four upper nodes in S^1, are summarized by the top summary node in S^3 which has 0 for property o and an indefinite value for property e. This is because the four upper nodes in S^1 have 0 for property o and different values for property e.

After compacting the irrelevant context by one node, it is given the property ic.

5.3.2 Intraprocedural statements

Intraprocedural statements are handled as in the algorithm in Chapter 4. The property ic is not affected by intraprocedural statements. It acts as a permanent mark distinguishing the node that represents the irrelevant calling context from other nodes.

5.3.3 Return statements

Given a structure S^x at the exit-site of procedure `pr` *and* a structure S^c at a call-site to procedure `pr`, the algorithm generates a structure S^r at the return-site by (a) "replacing" the representation of the irrelevant calling context, in S^x by the more detailed representation of S^c, and (b) popping the top activation record (This is done like in the algorithm presented in Chapter 4.)

Formally, replacing the representation of the irrelevant calling context, is done by applying two new operations *project* and *combine* defined below.

Definition 5.1 (Projection of a strucutre) *For a structure $S = \langle U, \iota \rangle$ and a set of nodes $W \subseteq U$, the projection of S on W, denoted by $S|_W$ is the structure S' where $U^{S'} = W$ and $p^{S'}(u_1, \ldots, u_k) = p^S(u_1, \ldots, u_k)$.*

Projecting a *3-valued logical strucutre* S on a set $W \subseteq U^S$, results in a structure $S|_W$ which is "smaller" than S, it contains a subset of the nodes in U^S

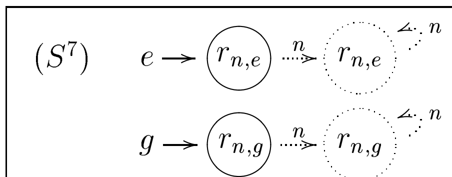

Figure 5.2: An Example for the *project* operator. S^7 is the projection of S^1 on the four nodes at the top of the diagram.

(assuming $W \neq U^S$), but the properties of the nodes in its universe matches their properties in S. For example, projecting S^1 shown in Figure 5.1 on the four upper nodes results in the structure S^7 shown in Figure 5.2. Notice that the universe of S^7 contains only the four nodes shown, in particular it does not contain activation records.

Definition 5.2 (Structure combination) *For two logical structures S^1 and S^2 over the same vocabulary $\mathcal{P}$. The combination of S^1 and S^2, denoted combine(S^1, S^2) is the structure S where*

$$U^S = \{u.1 : u \in U^1\} \bigcup \{u.2 : u \in U^2\}$$
$$p^S(u_1, \ldots, u_k) = \begin{cases} [\![p(v_1, \ldots, v_k)]\!]^{S^1}(v_1 \mapsto w_1, \ldots, v_k \mapsto w_k) & u_1 = w_1.1, \ldots, u_k = w_k.1 \\ [\![p(v_1, \ldots, v_k)]\!]^{S^2}(v_1 \mapsto w_1, \ldots, v_k \mapsto w_k) & u_1 = w_1.2, \ldots, u_k = w_k.2 \\ 0 & \textit{otherwise} \end{cases}$$

The *combine* operation takes two structures S^1 and S^2, and generates one. Each node in the resulting structure "stands" for exactly one node from either U^{S^1} or U^{S^2}, and has the same properties as this node. Figure 5.3 gives an example for combination of structures. The names of the nodes are written in subscripts to demonstrate the renaming of elements.

The algorithm replaces the representation of the irrelevant calling context in a structure S^x at the exit-site, with the irrelevant context representation in a structure at a call-site of `pr`, S^c by: (i) projecting S^x on the set of all nodes that do not have the *ic* property, effectively removing the irrelevant context node from the structure, (ii) projecting S^c on the set of irrelevant nodes for procedure `pr`, and (iii) finally, combining the resulting structures, thereby, obtaining the representation of the irrelevant nodes at the call-site.

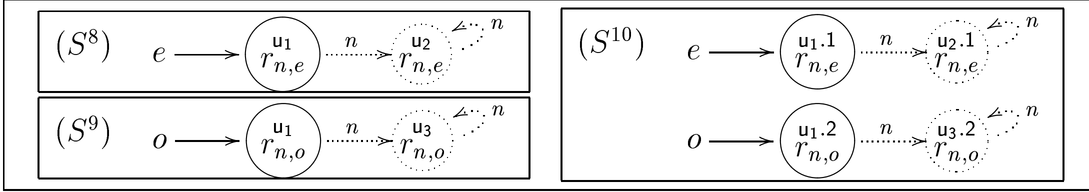

Figure 5.3: An example for the combine operator $S^{10} = combine(S^8, S^9)$.

This is captured formally by,

$$S = combine(S^x|_{\{u:u \in U^{S^x}, [\![ic(u)]\!]^{S^x}=0\}}, S^c|_{\{u:u \in U^{S^c}, [\![r_{pr}(u)]\!]^{S^c}=0\}})$$

For example, S^6 shown in Figure 5.1, is the result of combining S^7 shown in Figure 5.2 which is the projection of S^1 on the set of irrelevant nods for `set_insert` with S^5 shown in Figure 5.1 projected on all the nodes except the node with the ic property.

5.4 Discussion

The analysis presented in this chapter improves the one presented in Chapter 4 by using a less informative representation of irrelevant calling contexts. The loss of information enables the algorithm to save space in the analysis of a procedure body, but might lead to a loss of precision: combining every structure at the exit-site with every structure at the entry-site might create superfluous structures at the return-site. It is possible to reduce this problem by combining only "compatible" structures: a structure S^c at the call-site is compatible with a structure S^x at the exit-site if compacting the irrelevant context of S^c results in a node with the same properties as the irrelevant context node in S^x.

The analysis can even adopt a more "aggressive" compaction scheme. Instead of minimizing the information loss (by keeping definite values of predicates for the irrelevant context node), the analysis can set the values of all predicates to be unknown at this node. This way requires less structures in the analysis of a procedure body, but prohibits combining of only "compatible" structures.

The most interesting aspect of the analysis is that it varies the abstractions at different program points. The reason that the analysis can replace

the less informative representation of the irrelevant context at the exit site with the more detailed one at the call-site, is that that a procedure cannot modify irrelevant portions of the memory, and that the values of the instrumentation predicates of a node u do not change, even if properties of nodes which are not connected to it change.

A more ambitious approach would be to treat the heap as a directed graph, and to choose as the irrelevant context all the elements that are not reachable from the variables the procedure refers to. This approach requires a more sophisticated *combine* operator, which we do not have.

For non recursive procedures, local variables and parameters can be handled by rewriting the program. Recursive calls can be handled by considering heap nodes reachable from invisible variables in the relevant context. We do not know if this will save a lot of space.

Chapter 6

Analysis of Programs Manipulating Abstract Data Types

6.1 Overview

We have also investigated a more efficient solution for programs which manipulate the heap in a "more controled way". The idea is that the application program uses abstract data types, ADTs, that are only modified using a fixed set of interface procedures. In particular, it does not directly access the content of the heap. We also make certain assumptions on the ADT implementation explained in the sequel.

Because of time constraints and because we wanted to handle *existing* software, in this thesis, we only briefly studied this direction. In the rest of this Chapter we demonstrate the application of this idea to a queue data structure and then describe our initial experience of applying the same ideas to the LEDA [Näh95, MN99] linked list data structure implementation.

6.2 Analyzing a queue

We demonstrate the application of our idea for the analysis of queue manipulating programs. A *queue* data structure is composed of an object of type *Queue* declared in Figure 6.1 and a related list. A *Queue* is comprised of two pointer components: `hd`, the queue head, and `tl`, the queue tail. The

two components point to the list which contains the queue elements. The `hd`-component points to the head of the list, and the `tl`-component points to its last element. The procedures declared in Figure 6.1 comprise the queue interface: a queue is created by invoking the procedure `createQ`. The procedure dynamically allocates a new Queue object and sets its two pointer components to NULL, indicating that the queue is empty. The queue can be modified by invoking `enqQ` which inserts an element at the end of the queue, or by invoking `deqQ` which removes the element at the head of the queue. Information regarding the state of the queue can be obtained by invoking the procedure `emptyQ` which checks if the queue is empty or not, or by invoking the procedure `headQ` which returns the element stored at the head of the queue. The procedures `emptyQ` and `headQ` do not modify the queue. When the queue is no longer needed, it is finalized by invoking `destroyQ`. The implementation of these interface procedures is shown in Figure 6.2.

Except for procedure `createQ`, which has no parameters, all other interface procedures receive a pointer to a Queue as a parameter. They all require that this pointer would not be NULL. In addition, the procedures `headQ` and `deqQ` require that the queue would not be empty, whereas procedure `destroyQ` requires that the queue would be empty. Our purpose is to verify that a program (which manipulates queues only using the interface procedures) does not violate these preconditions, at a moderate cost.

6.2.1 A cheaper representation of the memory state

The main idea is to use a cheaper representation, in terms of space, of the program memory state: Instead of representing every Queue object and every list element by an individual as we did in the previous chapters, we represent an entire queue data structure by one individual. The predicates that we use to record properties of an entire queue data structure are given in Table 6.1. The predicate $x(v)$ captures the property that the queue v can be manipulated by invoking an interface procedure on the variable `x`. The predicate $E(v)$ records the state of the queue: it holds when v is an empty queue.

Example 6.2.1 Consider the program shown in Figure 6.4, at l_3 the queue pointed to by `x` contains two elements: 4 and 2. Figure 6.3(a) graphically depicts the memory state at this label. The leftmost rectangle is a Queue object. Its `hd`-component points to the list element with the data value 4, and its `tl`-component points to the list element with the data value 2.

```
/* Queue.h */
#include "List.h"

typedef struct {
  L hd;
  L tl;
} Queue;
typedef Queue *Q; /* Q is a pointer to a Queue */

typedef enum { FALSE,TRUE } bool;

/* Returns a pointer to a new empty Queue. */
Q createQ();
/* Inserts v at the tail of q. */
void enqQ(Q q, int v);
/* Deletes the element at the head of q. */
void deqQ(Q q);
/* Returns the element at the head of q. */
int headQ(Q q);
/* Returns TRUE if q is empty, FALSE otherwise. */
bool emptyQ(Q q);
/* Destroys q. */
void destroyQ(Q q);
```

Figure 6.1: A declaration for the Queue type and the interface procedures that manipulate it.

```
Q createQ() {
      Q q;
      q = (Q) malloc(sizeof(*q));
      q->hd = NULL;
      q->tl = NULL;
      return q;
}
```

```
void destroyQ(Q q) {
      L h;
      if (q == NULL) error();
      h = q->hd;
      if (h != NULL) error();
      free (q);
}
```

```
void enqQ(Q q, int v) {
      L t,e;
      if (q == NULL) error();
      e = (L) malloc(sizeof(*t));
      e->n = NULL; e->d = v;
      t = q->tl;
      if (t == NULL)
             q->hd = e ;
      else
             t->n = e ;
      q->tl = e ;
  }
```

```
void deqQ(Q q) {
      L h,e;
      if (q == NULL) error();
      h = q->hd;
      if (p == NULL) error();
      e = p->n;
      q->hd = e;
      if (e == NULL)
             q->tl = NULL;
      free(h);
  }
```

```
bool emptyQ(Q q) {
      L h;
      if (q == NULL) error();
      h = q->hd;
      if (h == NULL)
             return T;
      return F;
}
```

```
int headQ(Q q) {
      L h;
      int v;
      if (q == NULL) error();
      h = q->hd;
      if (h == NULL) error();
      v = h->d;
      return v;
}
```

Figure 6.2: An implementation of a Queue. Whenever a precondition is violated procedure `error` is called.

Predicate	Intended Meaning
$x(v)$	The Queue object of queue v is pointed to by a global variable `x`.
$E(v)$	The list of queue v is empty.

Table 6.1: The predicates used in the analysis of queue manipulating programs. There is a separate predicate x for every variable x which points to a Queue.

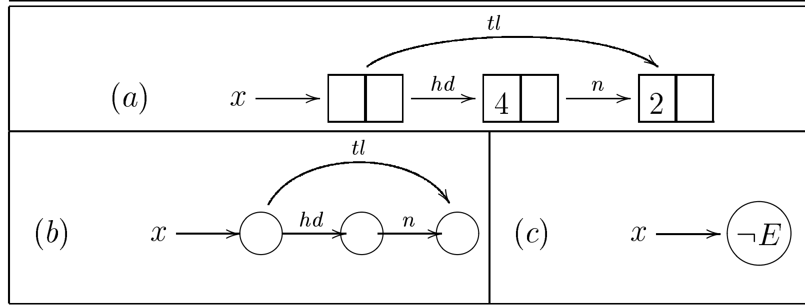

Figure 6.3: Representation of memory states by a *2-valued logical strucutre* as suggested at Chapter 4, compared to the representation suggested in this chapter.

The `n`-component of the list element with data value 4, points to the list element with the data value 2. The edge from the label x to the rectangle representing the Queue object indicates that the Queue element is pointed to by the variable `x`. (b) and (c) show two graphical representations of this memory state. (b) is the representation similar to the one given in previous chapters and (c) is new here, it is a more abstract memory representation. (b) represents every memory cell by a node. The leftmost node represents the Queue object, and the other two nodes represent the list elements. The edges labeled hd, tl and n, represent the value of the `hd`, `tl` and `n` components respectively. In contrast, (c) represents the entire queue by one node with extra "state" information. The edge from the label x to the node indicates that the Queue element of the queue is pointed to by variable `x`. The fact that the queue is not empty is indicated by $\neg E$ inside that node. This means that this node does not have the property E.

```
Q x;

void main() {
 l0 : x = createQ();
 l1 : enqQ(x,4);
 l2 : enqQ(x,2);
 l3 :
}
```

Figure 6.4: A queue-manipulating program.

6.2.2 Representing the effect of a procedure by a finite state machine

The effect of invoking an interface procedure on a queue is reflected by a (possible) change of the queue properties. Inserting an element into an empty queue results in a non-empty queue. Deleting an element from a non empty queue, may yield an empty queue, when the queue contains one element, or leave the queue non-empty otherwise. In all other cases the queue properties do not change. Note that the only property that changes is the "emptiness" property. The "pointed-to by variable `x`" property does not change by an invocation of an interface procedure. The difference between these properties is that the "emptiness" captures an "internal state" of the data structure, while the "pointed-to by a variable" property captures the calling context, which neither effects any interface procedure behaviour, nor it is effected by it.

The effects of an invocation of an interface procedure is represented compactly by a non-deterministic Finite State Machine (FSM). The FSM states represent different values of the "internal state" properties , and the FSM transitions represent the changes of these values due to an interface procedure invocation. The FSM for the queue data structure is shown in Figure 6.5. It has four states: E, $\neg E$, er, and na depicted as ellipses. The FSM states represents the possible "internal states" of a queue: the state E is for an empty queue, and the state $\neg E$ is for a non-empty queue. The error state er and the not-allocated state na are explained later. The transitions of the

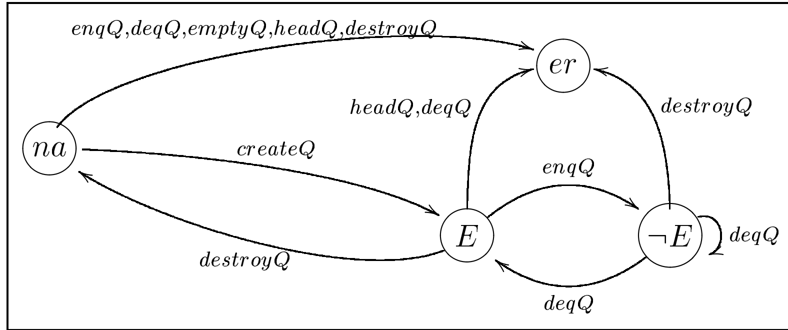

Figure 6.5: A non-deterministic FSM which represents the possible changes of the queue properties by invoking an interface procedure.

FSM are depicted as edges labeled with procedure names. They represent a change in the queue "internal state" due to a procedure invocation, e.g., the edge from state E to state $\neg E$ represents the change of the queue from being empty to being not empty due to an insertion of an element. The lack of an edge labeled with a procedure name emanating from a state, implicitly represents that the queue properties do not change due to the procedure invocation. For example, the lack of an edge labeled with procedure `enqQ` name emanating from state $\neg E$, indicates that a non-empty queue stays non-empty in case an element is inserted into it. The error state er is a special state which represents a queue in an "unknown" state. It is reached when a procedure is invoked without a necessary precondition. For instance, the edge labeled `headQ,deqQ` from state E to the error state, indicates that removing an element from an empty queue is an error. The "not-allocated" state na represents a queue that does not exist; since a queue is comprised of dynamically allocated objects, it exists only after it is allocated (by procedure `createQ`) and before it is destroyed (by procedure `destroyQ`). Invoking any interface procedure on a queue that does not exist (either on a null pointer or on a queue which was destroyed) yields an error. This is represented by the edge from state na to state er. The edge labeled `createQ` emanating from state na indicates that when a queue is created, it is empty.

6.2.3 The analysis

The analysis algorithm annotates a program point l_i with a set of structures that represent all the possible memory states that may arise before the execution of the statement at l_i. The algorithm utilizes the queue FSM to find the effect of a procedure invocation on the memory state as demonstrated in the following example.

Example 6.2.2 Figure 6.6 contains the result of the analysis applied to the program shown in Figure 6.4.

- Before l_0, no statement is executed, and in particular no memory element is allocated. Thus, the (one) structure that represents the memory state at this point has an empty universe.

- The statement `x = createQ()` at l_0, allocates a queue and sets `x` to point to it. Thus, the structure at l_1 results by adding a node to the structure at l_0. This node gets the property E, since the FSM edge emanating from the not-allocated state points to the empty state. In addition this node has the property x since the return value of the procedure, which points to the new queue, is assigned to `x`.

- The statement `enqQ(x,4)` at l_1 inserts 4 into the queue pointed to by `x`. Thus, the structure at l_2 results from the structure at l_1 by removing the E property from the node which has the x property. The algorithm performs this action since the queue `x` points to at l_1 is empty, and the FSM edge labeled `enqQ` emanating from the empty state leads to the non-empty state.

- The statement `enqQ(x,2)` at l_2 inserts 2 into the queue pointed to by `x`. At l_2, `x` points to a non-empty queue. Thus, the structure at l_3 is identical to the structure at l_2, since invoking `enqQ` on a non-empty queue results in a non-empty queue.

6.3 Experience with LEDA

We applied FSMs to analyze singly linked lists data structure implementation in LEDA [MN99]. LEDA is a library of generic data structures written

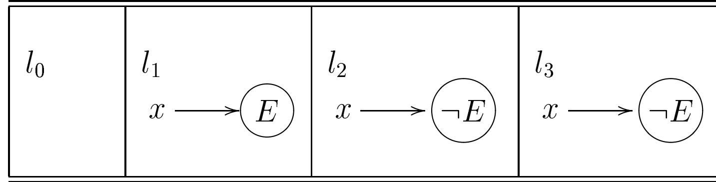

Figure 6.6: The representation of the memory state at each point of the queue-manipulating program shown in Figure 6.4.

in C++. LEDA implementation of singly linked lists is similar to the implementation of the queue presented in the previous section: A list is comprised of a SLIST object, which like Queue object contains pointers to the head and the tail of a list of nodes. However, LEDA list supports more interface procedures then queue, e.g., an element can be inserted at the head of the list too (pushed).

We wanted to compare the results of the FSM based analysis with the analysis presented in Chapter 4. In order to do this, we rewrote in C a specific instance of the list which manipulates integers. We analyzed a program which uses a queue implemented by two list that function as stacks (([MN99, p.60] shows such a data structure). The analysis of this program by a suitable variant of the algorithm presented in Chapter 4, requires 5777 structures and takes 392.4 seconds, whereas an analysis similar to the one presented in this chapter (which distinguishes between four "internal states": an empty list, a list containing one element, a list containing two elements and a list of three or more elements), requires 442 structures and takes only 4.5 seconds. The two analyses verified that the program does not violate the interface procedures preconditions. The experiment was carried on a Pentium II 233 Mhz machine running Windows 2000 with JDK 1.2.2.

6.4 Discussion

The analysis of LEDA list is a bit more complicated than the analysis of the queue. Unlike the queue interface procedures, some of the LEDA list interface procedures gets two list parameters. We modeled them too using FSMs. We found that using one FSM for each procedure is not good enough. Instead, we used two FSMs: one for the case the two arguments are aliased

and one for the case they are not. Otherwise the results are too conservative. While modeling the procedures, we found that the LEDA procedure which concatenates two linked list is flawed: when a list is concatenated to itself, a cycle is created which leads to an error later. This bug was reported to the LEDA development team.

The LEDA experiment highlights some of our limitations: LEDA is generic, it allows for a data structure to contain elements which are data structures themselves, we do not support this. Another limitation is the fact that we do not support iterators. Iterators enable the program to access parts of the information stored in a data structure not through an interface procedure. This allows for the program to modify this information in an "uncontrolled way" which is not supported. Currently we can analyze only programs which manipulate a bounded number of data structures using "shallow pointers": pointers that never point to an heap object which is pointed to by another heap element. In particular, the data structures cannot share elements, otherwise manipulating one data structure might affect another. Finally, perhaps the main limitation of this approach, is that it requires an a priori creation of the FSMs for the ADT implementation which may be hard, if not impossible for a practical ADT implementation. We made a little progress in the creation of the FSM in our example by using the analysis given in Chapter 4.

The advantage of this analysis is its simplicity and the ability to analyze programs that manipulate data structures for which we do not have a more general "expensive" analysis. Furthermore, the ADT implementation code is not analyzed.

In our queue example, we use the state na to represent a queue which does not exist. This state represents a queue which is either not created yet or destroyed. Thus, we assume that a pointer is nulled when the queue it points to is destroyed. This is not faithful to C semantics which leaves dangling pointers. It can be easily fixed by adding a state *freed* to distinguish between the *not-created* case and the *freed* case.

Chapter 7

Prototype Implementation

A prototype of the iterative algorithm sketched in Section 4.3 was implemented for a small subset of C. The main goal has been to determine if the results of the analysis are useful before trying to scale the algorithm to handle arbitrary C programs. In theory, the algorithm might be overly conservative and yield many indefinite values. This may lead to many "false alarms". For example, the algorithm might have reported that every program point possibly leaked memory, performed a NULL-pointer dereference, etc. Fortunately, in Section 7.1 we show that this is not the case for the C procedures analyzed.

As explained in Section 4.3, the algorithm does not explicitly apply the operational semantics to each of the (potentially infinite) structures represented by a three-valued structure S. Also, it does not explicitly apply it to S either, for this might not be precise enough. For example, evaluating the update formula for *top* on the structure $S_{4.1}$ shown in Figure 4.1 would give top' an indefinite value at the stack summary node. This is because the node representing the activation record of the caller is summarized with other activation record nodes in $S_{4.1}$. To overcome this problem, before applying the operational semantics, the algorithm replaces S by a set of *3-valued logical strucutre* $\widehat{S}$ that (i) any structure in $\widehat{S}$ can be embedded in S and (ii) any *2-valued logical strucutre* that is embedded in S can be embedded in one of the structures in $\widehat{S}$. Thus, together, all the structures in $\widehat{S}$ represent all the memory states that S represents. Unlike S, in any of the structures in $\widehat{S}$ the update formulae of the core predicate evaluates to a definite value. Finding $\widehat{S}$ in the framework of [SRW99] is achieved via an operation called *Focus*.

The *Focus* operation might generate inconsistent structures. For exam-

ple, applying focus to $S_{4.1}$ adds into $\widehat{S_{4.1}}$ a structure in which the current activation record has no previous (i.e. the update formula for the predicate *top* evaluates to 0), which is impossible. Such structures are excluded by applying *Coerce*, an operation that performs a series of similar "sanity checks" guaranteeing that exclusion of impossible structures. *Coerce* can also increase the precision by lowering indefinite into definite values. More about *Focus* and *Coerce* can be found in [SRW99].

Instead of calculating the instrumentation predicate values at the resulting structure by their defining formulae, which may be overly conservative, predicate-update formulae for instrumentation predicates are used. Our implementation uses a variant of the predicates that appear in Table 4.1: The visible local variables are represented as global variables.

In addition we found that the analysis yield quite imprecise value for the $r_{n,\widehat{x}}(v)$ predicate. Instead, for every procedure `pr` we keep a predicate $r_{n,\widehat{x,pr}}(v)$ which records the reachable elements from the most recent invisible copy of `x` (the copy of `pr` last invocation). Because of this, our prototype implementation handles only programs with linear recursion.

The algorithm has been implemented using a 3-valued logic analysis system called TVLA [LAS00] (for **T**hree-**V**alued-**L**ogic **A**nalyzer). The inputs to the system are the procedure's control flow graph, and logical formulae describing the operational semantics of each statement and condition. The system iteratively computes a set of 3-valued structures at every program point. It is quite powerful but slow, and only supports analysis specified using low level logical formulae. Therefore, we implemented a frontend that generates TVLA input from a program in a subset of C. Although, TVLA supports only intraprocedural analysis our explicit manipulation of the activation record stack allows our frontend to treat call and return statements in the same way that intraprocedural statements are handled. The instrumentation predicates that relate heap without sacrificing precision in many recursive procedures.

Our front end also performs certain minimal optimizations not described here.

7.1 Empirical results

The analyzed C programs together with the space used and the running time are listed in Table 7.1. All our experiments were carried on a Pentium II 233 Mhz

Proc.	**Description**	**# of Structs**	**Time (secs)**
create	creates a list.	219	5.91
delall	frees the entire list	139	13.10
insert	creates and inserts an element into a sorted list	344	38.33
delete	deletes an element from a sorted list	423	31.69
search	searches an element in a sorted list	303	8.44
app_r	adds one list to the end of another	326	42.81
rev	the running example (non recursive append)	829	105.78
rev_r	the running example (with recursive append)	2285	1208.80
rev_d	reverses a list with destructive updates	429	45.99

Table 7.1: The total number of 3-valued structures that arise during analysis and running times for the recursive procedures analyzed. The procedures are given in Section A.4.

machine running Windows 2000 with JDK 1.2.2. The analysis verified that indeed these procedures always return a linked list and contain no memory leaks and NULL-pointer dereferences. Verifying the absence of memory leaks is quite challenging for these procedures since it requires information about invisible variables as described in Section 4.1.

Chapter 8

Conclusions and Future Work

We present a novel interprocedural shape analysis algorithm for programs that manipulate linked lists. The algorithm is more precise than existing shape analysis algorithms described in [JM81, CWZ90, SRW98] for recursive programs that destructively update the program store. The precision of our algorithm can be attributed to the properties of invisible instances of local variables that it tracks. Previous algorithms [JM81, CWZ90] either did not handle the case where multiple instances of the same local variable exist simultaneously, or only represented their potential values [SRW98]. As we have demonstrated, in the absence enough information about the values of local variables, an analysis must make overly conservative assumptions. These assumptions lead to imprecise results and performance deteriorates as well, since every potential value of a local variable must be considered.

We have identified several properties of local variables that help to overcome the unavoidable imprecision involved in the summarization of local variable's values. Tracking these properties enables the analysis to recover the values of the invisible variables at a return statement quite precisely in many cases. Particularly important seems to be the sharing properties of stack variables. Existing shape-analysis algorithms [JM81, CWZ90, SRW98] rely on the representation of the program store to infer such properties, but only record sharing properties of the heap. This approach may suffice in cases when the number of local variables is fixed, however in the presence of recursive procedure calls, different incarnations of local variables may point to the same heap cell. Keeping track of both the sharing properties of local variables and sharing properties of heap elements turned out to be key for the precision of our algorithm.

A prototype of the algorithm has been implemented.

Our algorithm was developed within the parametric framework descried in [SRW99, LAS00]. That framework allows the generation of intraprocedural shape-analysis algorithms based on an appropriate specification. We show that their framework can generate quite precise interprocedural algorithms too.

We follow the approach suggested in [JM82, Deu90] and summarize activation records in essentially the same way that linked list elements are summarized. By representing the call site in each activation record the analysis algorithm is capable of encoding the calling context, too. This approach bears some similarity to the *call-string* approach of [SP81], since it avoids propagating information to return sites that do not match the call site of the current activation record. In our case there is no need to put an arbitrary bound on the "length" of the call-string, the bounded representation is achieved indirectly by the summarization of activation records.

In our bounded representation, heap elements that are pointed to by invisible variables may eventually be summarized. However, if after a return statement, a summary node represents heap cells that are pointed to by visible local variables, imprecision is likely to arise. Thus, the ability to *materialize* a non–summary node out of a summary node is necessary for the precision of our algorithm [SRW98].

Our *compaction* approach for scaling the algorithm reminds the use of *frame axioms* in program correctness proofs [GL80]. A frame axiom defines an invariance of the procedure. We rely on the fact that a portion of the memory that the procedure cannot refer to, cannot be modified by it. Thus, our solution can be viewed as a special case of an automatically determined frame axiom. An even more ambitious approach, is to summarize the effect of each procedure by a "summary function". Such an approach is taken in [CRL99, WL95] for a points-to analysis. A Points-to analysis finds aliasing between pointer variables. However the relation between heap allocated objects is not tracked.

Besides shape analysis there are other methods to handle programs with pointer including [HN90, GH96, Deu94, CBC93]. These methods are incomparable to our method. The power of shape analysis stems from the ability to handle arbitrary programs and to conduct strong updates, even for example, when the program manipulates a cyclic list. The power Deutsch's method is the ability to precisely handle recursive traversal of linked data structures.

Our approach for analyzing program manipulating ADTs is similar to the

typestate approach introduced in [Str83, SY86]. A *typestate* represents the set of runtime states of a variable at each program point, and finite state machines to represent the effect of procedure calls (and operations) on the variable state. However, they do not handle aliasing between variables and require runtime checks at program points where a data structure is updated and alias might occur. Their work sheds interesting light on the extension possibilities of our work: They allow for a data structure to contain another data structure (i.e., a table of records), however before an "internal" data structure is updated it must be either detached from the container data structure, or a runtime check must verify that the internal data structure cannot be accessed by any other program variable. We conjecture that that our method can be extended to handle nested ADTs by using a similar detachment operation. Such a scheme may allow us to analyze typestates without runtime checks.

8.1 Current limitations and future work

So far, our technique (and our implementation) analyzes small programs in a "friendly" subset of C. We plan to extend it to a larger subset of C, and to experiment with scaling it up to programs of realistic size. One possible way involves first running a cheap and imprecise pointer-analysis algorithm, such as the flow-insensitive points-to analysis described in [Ste96], before proceeding to our quite precise but expensive analysis. An implementation of our *compaction* algorithm would allow us to test the actual benefits of this technique.

We focused this research on linked lists, but, plan to also investigate tree-manipulation programs.

Finally, our analysis is limited by its fixed set of predefined "library" properties. This makes our tool easy to use since it is fully automatic and does not require any user intervention, e.g., a specification of the program. However, this is a limitation because the analyzer produce poor results for program in which other properties are the important distinctions to track.

Bibliography

[AW93] U. Assmann and M. Weinhardt. Interprocedural heap analysis for parallelizing imperative programs. In W. K. Giloi, S. Jähnichen, and B. D. Shriver, editors, *Programming Models For Massively Parallel Computers*, pages 74–82, Washington, DC, September 1993. IEEE Press.

[CBC93] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side-effects. In *Symp. on Princ. of Prog. Lang.*, pages 232–245, New York, NY, 1993. ACM Press.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Symp. on Princ. of Prog. Lang.*, pages 269–282, New York, NY, 1979. ACM Press.

[CRL99] Ramkrishna Chatterjee, Barbara G. Ryder, and William A. Landi. Relevant context inference. In ACM, editor, *POPL '99. Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of programming languages, January 20–22, 1999, San Antonio, TX*, ACM SIGPLAN Notices, pages 133–146, New York, NY, USA, 1999. ACM Press.

[CWZ90] D.R. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 296–310, New York, NY, 1990. ACM Press.

[Deu90] A. Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *Symp. on Princ. of Prog. Lang.*, pages 157–168, 1990.

[Deu94] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 230–241, New York, NY, 1994. ACM Press.

[DRS98] N. Dor, M. Rodeh, and M. Sagiv. Detecting memory errors via static pointer analysis. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98)*, pages 27–34, June 1998. Available at "http://www.math.tau.ac.il/∼ nurr/paste98.ps.gz".

[DRS00] N. Dor, M. Rodeh, and M. Sagiv. Checking cleanness in linked lists. In *SAS'00, Static Analysis Symposium*. Springer, 2000. Available at "http://www.math.tau.ac.il/∼ nurr".

[GH96] R. Ghiya and L.J. Hendren. Is it a tree, a dag, or a cyclic graph? In *Symp. on Princ. of Prog. Lang.*, New York, NY, January 1996. ACM Press. Available at "ftp://ftp-acaps.cs.mcgill.ca/pub/doc/papers/POPL96.ps.gz".

[GH98] R. Ghiya and L. Hendren. Putting pointer analysis to work. In *Symp. on Princ. of Prog. Lang.*, New York, NY, 1998. ACM Press.

[GL80] D. Gries and G. Levin. Assignment and procedure call proof rules, 1980.

[Hen90] L. Hendren. *Parallelizing Programs with Recursive Data Structures*. PhD thesis, Cornell Univ., Ithaca, NY, Jan 1990.

[HHN92] L. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and the transformation of imperative programs. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 249–260, New York, NY, June 1992. ACM Press.

[HN90] L. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Trans. on Par. and Dist. Syst.*, 1(1):35–47, January 1990.

[HPR89] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 28–40, New York, NY, 1989. ACM Press.

[JM81] N.D. Jones and S.S. Muchnick. Flow analysis and optimization of Lisp-like structures. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 4, pages 102–131. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[JM82] N.D. Jones and S.S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Symp. on Princ. of Prog. Lang.*, pages 66–74, New York, NY, 1982. ACM Press.

[LAS00] T. Lev-Ami and M. Sagiv. TVLA: A framework for Kleene based static analysis. In *SAS'00, Static Analysis Symposium*. Springer, 2000. Available at http://www.math.tau.ac.il/∼tla.

[LH88] J.R. Larus and P.N. Hilfinger. Detecting conflicts between structure accesses. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 21–34, New York, NY, 1988. ACM Press.

[MN99] K. Mehlhorn and S. Naher. *LEDA, A Platform for Combinatorical and Generic Computing*. Cambridge University Press, first edition, 1999.

[Näh95] S. Nähr. *The LDEA User Manual*, 1995. Available via anonymous ftp from ftp.mpi-sb.mpg.de.

[PCK93] J. Plevyak, A.A. Chien, and V. Karamcheti. Analysis of dynamic structures for efficient parallel execution. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, volume 768 of *Lec. Notes in Comp. Sci.*, pages 37–57, Portland, OR, August 1993. Springer-Verlag.

[SP81] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[SRW98] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *Trans. on Prog. Lang. and Syst.*, 20(1):1–50, January 1998.

[SRW99] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Symp. on Princ. of Prog. Lang.*, 1999. Available at "http://www.cs.wisc.edu/wpis/papers/popl99.ps".

[Ste96] B. Steensgaard. Points-to analysis in almost-linear time. In *Symp. on Princ. of Prog. Lang.*, pages 32–41, 1996.

[Str83] R. Strom. Mechanism for compile-time enforcement of security, 1983.

[Str92] J. Stransky. A lattice for abstract interpretation of dynamic (Lisp-like) structures. *Inf. and Comp.*, 101(1):70–102, Nov. 1992.

[SY86] R.E. Strom and S.A. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, SE-12(1):157–171, 1986.

[Wan94] E. Y.-B. Wang. *Analysis of Recursive Types in an Imperative Language.* PhD thesis, Univ. of Calif., Berkeley, CA, 1994.

[WL95] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for c programs, 1995.

[Zap99] Emilio Zapata. Automatic parallelization of irregular applications. In *SPA99*, 1999.

Appendix A

Appendix

A.1 Syntax of formulae

In this section , we define the syntax of first-order formulae with equality and transitive closure we use.

Definition A.1.1 *A* **formula** *over the* **vocabulary** $\mathcal{P} = \{p_1, \ldots, p_n\}$ *is defined inductively, as follows:*

Atomic Formulae *The* **logical literals 0**, **1**, *and* **1/2** *are atomic formulae with no free variables.*

For every predicate symbol $p \in \mathcal{P}$ *of arity* k, $p(v_1, \ldots, v_k)$ *is an atomic formula with free variables* $\{v_1, \ldots, v_k\}$.

The formula $(v_1 = v_2)$, *where* v_1 *and* v_2 *are distinct variables, is an atomic formula with free variables* $\{v_1, v_2\}$.

Logical Connectives *If* φ_1, φ_2 *and* φ_3 *are formulae whose sets of free variables are* V_1, V_2, *and* V_3 , *respectively, then* $(\varphi_1 \wedge \varphi_2)$, $(\varphi_1 \vee \varphi_2)$, $(\neg\varphi_1)$, *and* $(\varphi_1 ? \varphi_2 : \varphi_3)$ *are formulae with free variables* $V_1 \cup V_2$, $V_1 \cup V_2$, V_1, *and* $V_1 \cup V_2 \cup V_3$ *respectively.*

Quantifiers *If* φ *is a formula with free variables* $\{v_1, v_2, \ldots, v_k\}$, *then* $(\exists v_1 : \varphi)$ *and* $(\forall v_1 : \varphi)$ *are both formulae with free variables* $\{v_2, v_3, \ldots, v_k\}$.

Transitive Closure *If* φ *is a formula with free variables* V *such that* $v_3, v_4 \notin V$, *then* $(TC\ v_1, v_2 : \varphi)(v_3, v_4)$ *is a formula with free variables* $(V - \{v_1, v_2\}) \cup \{v_3, v_4\}$.

A formula is **closed** *when it has no free variables.*

A.2 Kleene's 3-valued semantics

In this section we define Kleene's 3-valued semantics for first-order formulae with transitive closure.

Definition A.2.1 *A* **3-valued interpretation** *of the language of formulae over $\mathcal{P}$ is a* **3-valued logical structure** *S, comprised of U^S, a set of* **individuals** *and interpretation p^S for every predicate symbol p of arity k to a truth-valued function:*

$$p^S\colon (U^S)^k \to \{0, 1, 1/2\}.$$

An **assignment** *Z is a function that maps free variables to individuals (i.e., an assignment has the functionality $Z\colon \{v_1, v_2, \ldots\} \to U^S$). An assignment that is defined on all free variables of a formula φ is called* **complete** *for φ. In the sequel, we assume that every assignment Z that arises in connection with the discussion of some formula φ is complete for φ.*

The **meaning** *of a formula φ, denoted by $[\![\varphi]\!]_3^S(Z)$, yields a truth value in $\{0, 1, 1/2\}$. The meaning of φ is defined inductively as follows:*

Atomic *For a logical literal $\boldsymbol{l} \in \{\mathbf{0}, \mathbf{1}, \mathbf{1/2}\}$, $[\![\boldsymbol{l}]\!]_3^S(Z) = l$ (where $l \in \{0, 1, 1/2\}$).*

For an atomic formula $p(v_1, \ldots, v_k)$,

$$[\![p(v_1, \ldots, v_k)]\!]_3^S(Z) = p^S(Z(v_1), \ldots, Z(v_k))$$

For an atomic formula $(v_1 = v_2)$,

$$[\![v_1 = v_2]\!]_3^S(Z) = \begin{cases} 0 & Z(v_1) \neq Z(v_2) \\ 1 & Z(v_1) = Z(v_2) \text{ and } sm^S(Z(v_1)) = 0 \\ 1/2 & \text{otherwise} \end{cases}$$

Logical Connectives *For logical formulae φ_1, φ_2, and φ_3*

$$\begin{aligned}
[\![\varphi_1 \wedge \varphi_2]\!]_3^S(Z) &= \min([\![\varphi_1]\!]_3^S(Z), [\![\varphi_2]\!]_3^S(Z)) \\
[\![\varphi_1 \vee \varphi_2]\!]_3^S(Z) &= \max([\![\varphi_1]\!]_3^S(Z), [\![\varphi_2]\!]_3^S(Z)) \\
[\![\neg\varphi_1]\!]_3^S(Z) &= 1 - [\![\varphi_1]\!]_3^S(Z) \\
[\![\varphi_1 ? \varphi_2 : \varphi_3]\!]_3^S(Z) &= \begin{cases} [\![\varphi_2]\!]_3^S(Z) & [\![\varphi_1]\!]_3^S(Z) = 0 \\ [\![\varphi_3]\!]_3^S(Z) & [\![\varphi_1]\!]_3^S(Z) = 1 \\ [\![\varphi_2]\!]_3^S(Z) & \begin{array}{l} [\![\varphi_1]\!]_3^S(Z) = 1/2 \\ [\![\varphi_2]\!]_3^S(Z) = [\![\varphi_3]\!]_3^S(Z) \end{array} \\ 1/2 & otherwise \end{cases}
\end{aligned}$$

Quantifiers *If φ is a logical formula,*

$$\begin{aligned}
[\![\forall v_1 : \varphi]\!]_3^S(Z) &= \min_{u \in U^S} [\![\varphi_1]\!]_3^S(Z[v_1 \mapsto u]) \\
[\![\exists v_1 : \varphi]\!]_3^S(Z) &= \max_{u \in U^S} [\![\varphi_1]\!]_3^S(Z[v_1 \mapsto u])
\end{aligned}$$

Transitive Closure *For $(TC\ v_1, v_2 : \varphi)(v_3, v_4)$,*

$$[\![(TC\ v_1, v_2 : \varphi)(v_3, v_4)]\!]_3^S(Z) = \max_{\substack{n \geq 1, u_1, \ldots, u_{n+1} \in U, \\ Z(v_3) = u_1, Z(v_4) = u_{n+1}}} \min_{i=1}^{n} [\![\varphi]\!]_3^S(Z[v_1 \mapsto u_i, v_2 \mapsto u_{i+1}])$$

We say that S and Z **potentially satisfy** *φ (denoted by $S, Z \models \varphi$) if $[\![\varphi]\!]_3^S(Z) = 1/2$ or $[\![\varphi]\!]_3^S(Z) = 1$. Finally, we write $S \models \varphi$ if for every Z: $S, Z \models \varphi$.*

The only nonstandard part of Definition A.2.1 is the meaning of equality (denoted by the symbol '$=$'). The predicate $=$ is defined in terms of the sm predicate and the "identically-equal" relation on individuals (denoted by the symbol '$=$'):[1]

- Non-identical individuals u_1 and u_2 are unequal (i.e., if $u_1 \neq u_2$ then $u_1 \neq u_2$).

[1] Note that there is only a small typographical distinction between the syntactic symbol for equality, namely '$=$', and the symbol for the "identically-equal" relation on individuals, namely '$=$'. Throughout the thesis, it should always be clear from the context which symbol is intended.

- A non-summary individual must be equal to itself (i.e., if $sm(u) = 0$, then $u = u$).
- In all other cases, we throw up our hands and return 1/2.

Notice that Definition A.2.1 could be generalized to allow many-sorted sets of individuals. This would be useful for modeling heap cells of different types; however, to simplify the presentation, we have chosen not to introduce this mechanism.

A.3 The meaning of programs statements

The meaning functions for program statements are defined as transformers from 2-valued structures to 2-valued structures. These transformers are defined by collection of first order formulae evaluated against the original structure. The value of every predicate is determined by a corresponding formula. The main idea is that if a structure S represents a set of memory states that arise before statement st, than a structure S' that represents the corresponding set of stores that arise after st can be obtained by evaluating a suitable collection of formulae that capture the semantics of st.

Formally, for every statement st, the new values of every predicate p are defined via a predicate-update formula φ_p^{st}

Definition A.3.1 *Let st be a program statement, and for every arity-k predicate p in vocabulary $\mathcal{P}$, let φ_p^{st} be the formula over free variables $v_1, \ldots, v_k$ that defines the new value of p after st. Then, applying the operational semantics of st to a structure S yields a structure S' where $U^{S'} = U^S$ and*

$$p^{S'}(u_1, \ldots, u_k) = [\![\varphi_p^{st}]\!]_3^S([v_1 \mapsto u_1, \ldots, v_k \mapsto u_k])$$

Note that although the new values of the predicates is defined in 3-value logic, when applied to *2-valued logical strucutre* the resulting structure is a 2-valued logical structure. In particular, the semantics can be applied to 3-valued logical structures too.

A.3.1 Intraprocedural statements

Table A.1 lists the predicate-update formulae that define the semantics of the six kinds of intraprocedural statements that manipulate data structures defined by the `List` data type given in Figure 1.1. They are a variant of the ones in [SRW99]. Table A.1 lists a predicate-update formula φ_p^{st} only if predicate p is affected by the execution of the statement. For any unchanged predicate q, the predicate-update formula is "$\varphi_q^{st}(v_1, v_2, \ldots) = q(v_1, v_2, \ldots)$". To simplify the presentation, in Table A.1 (and elsewhere) we break each occurrence of `y->n = t` into two statements: `y->n = NULL`, followed by `y->n = t`, so that in the predicate-update formulae for `y->n = t` we can assume that `y->n == NULL`. Also, for simplicity, we define the semantics of statements involving a

dereference of a variable or a dynamic memory allocation or disposal only for global variables. We assume that such statements involving local variables were replaced by a series of statements involving temporary global variables. The third column in Table A.1 indicates the condition under which the formula is defined.

Definition A.3.1 does not handle statements that dynamically allocate or dispose memory (`x = malloc()` and `free(x)`) because the universe of the structure produced by $[\![st]\!](S)$ is the same as the universe of S. Instead, for these statements we use modified definitions of $[\![st]\!](S)$ given below.

For allocation statements we use the modified definition of $[\![st]\!](S)$ given in Definition A.3.2, which first allocates a new individual u_{new} (representing the allocated list element), and then invokes predicate-update formulae in a manner similar to Definition A.3.1.

Definition A.3.2 *Let* $st \equiv \texttt{x} = \texttt{malloc}()$ *and let* $new \notin \mathcal{P}$ *be a unary predicate. For every* $p \in \mathcal{P}$, *let* φ_p^{st} *be a predicate-update formula over vocabulary* $\mathcal{P} \cup \{new\}$. *Then, applying the operational semantics of* $st \equiv \texttt{x} = \texttt{malloc}()$ *to a structure* S , *yields a structure* S'' *defined as follows: First let* S' *be a structure with a new individual,* u_{new} *not in* U^S, *i.e.* $U^{S'} = U^S \cup \{u_{new}\}$ *and a new unary predicate* $new(v)$, *which holds only for* u_{new}, *i.e.,*

$$p^{S'}(u_1, \ldots, u_k) = \begin{cases} 1 & p = new \text{ and } u_1 = u_{new} \\ 0 & p = new \text{ and } u_1 \neq u_{new} \\ 0 & p \neq new \text{ and there exists } i, \\ & 1 \leq i \leq k, \text{ such that } u_i = u_{new} \\ p^S(u_1, \ldots, u_k) & \text{otherwise} \end{cases}$$

Now define $[\![x = malloc()]\!](S) = S''$ *where* $U^{S''} = U^{S'}$ *and*

$$p^{S''}(u_1, \ldots, u_k) = [\![\varphi_p^{st}]\!]_3^{S'}([v_1 \mapsto u_1, \ldots, v_k \mapsto u_k])$$

In Definition A.3.2, S' is created from S as follows: (i) $new(u_{new})$ is set to 1, (ii) $new(u_1)$ is set to 0 for all other individuals $u_1 \neq u_{new}$, and (iii) all predicates are set to 0 when one or more arguments is u_{new}. The predicate-update operation in Definition A.3.2 is very similar to the one in Definition A.3.1 after S' has been set.

When a procedure executes a $st \equiv$ `free(x)` statement it removes a list element from the program store. The operational semantics of st is defined

st	φ_p^{st}	Condition
`x = NULL`	$\varphi_x^{st}(v) \stackrel{\text{def}}{=} \mathbf{0}$ $\varphi_x^{st}(v_1, v_2) \stackrel{\text{def}}{=} x(v_1, v_2) \land \neg top(v_1)$	global `x` local `x`
`x = t`	$\varphi_x^{st}(v) \stackrel{\text{def}}{=} t(v)$ $\varphi_x^{st}(v) \stackrel{\text{def}}{=} \exists v_1 : top(v_1) \land t(v_1, v)$ $\varphi_x^{st}(v_1, v_2) \stackrel{\text{def}}{=} (\neg top(v_1) \land x(v_1, v_2)) \lor$ $(top(v_1) \land t(v_2))$ $\varphi_x^{st}(v_1, v_2) \stackrel{\text{def}}{=} (\neg top(v_1) \land x(v_1, v_2)) \lor$ $(top(v_1) \land t(v_1, v_2))$	global `x` global `t` global `x` local `t` local `x` global `t` local `x` local `t`
`x = t->n`	$\varphi_x^{st}(v) \stackrel{\text{def}}{=} \exists v_1 : t(v_1) \land n(v_1, v)$	global `x` global `t`
`x->n = NULL`	$\varphi_n^{st}(v_1, v_2) \stackrel{\text{def}}{=} n(v_1, v_2) \land \neg x(v_1)$	global `x` global `t`
`x->n = t` (assuming: `x->n == NULL`)	$\varphi_n^{st}(v_1, v_2) \stackrel{\text{def}}{=} n(v_1, v_2) \lor (x(v_1) \land t(v_2))$	global `x` global `t`
`x = malloc()`	$\varphi_x^{st}(v) \stackrel{\text{def}}{=} new(v)$ $\varphi_{heap}^{st}(v) \stackrel{\text{def}}{=} heap(v) \lor new(v)$	global `x`
`free(x)`	$\varphi_x^{st}(v) \stackrel{\text{def}}{=} 0$	global `x`

Table A.1: The predicate-update formulae defining the operational semantics of the call and return statements for the core predicates for `List`. `x` and `t` are variables in `P`Var.

by first applying the predicate update formulae, and then extracting the individual representing the released list element from the universe. Extracting the individual is done by the projection operation defined in Definition 5.1.

Definition A.3.3 *Let* $st \equiv$ `free(x)` *For every* $p \in \mathcal{P}$, *let* φ_p^{st} *be a predicate-update formula over vocabulary* $\mathcal{P}$. *Then, applying the operational semantics of* $st \equiv$ `free(x)` *to a structure* S , *yields a structure* $S' = S''|_{\{u \in U^S : x^S(u)=0\}}$ *where* $U^{S''} = U^S$ *and*

$$p^{S''}(u_1, \ldots, u_k) = [\![\varphi_p^{st}]\!]_3^S([v_1 \mapsto u_1, \ldots, v_k \mapsto u_k])$$

Thus, free is obtained by first evaluating the update formulae and then maintaining elements which are not originally pointed-to by `x`. Notice that this semantics is unfaithful to the C semantics which leaves dangling pointers. For simplicity we choose not to model such a behaviour, and require that no pointer other than `x` points to the freed cell.

A.3.2 Interprocedural statements

The meaning of interprocedural statements is defined by predicate update formulae too. Table A.2 lists the predicate update formulae for the interprocedural statements. Similarly to the intraprocedural statements that allocate or dispose dynamic memory, interprocedural statements modify the universe.

Definition A.3.4 is very similar to Definition A.3.2 but handle a procedure invocation statement. As in Definition A.3.2, Definition A.3.4 first allocates a new individual (representing the activation record of the invoked procedure), and then invokes predicate-update formulae. To simplify the presentation we assume that parameter passing is handled explicitly (i.e. procedures are parameterless)

Definition A.3.4 *Let* $st \equiv label:$ `call f()` *and let* $new \notin \mathcal{P}$ *be a unary predicate. For every* $p \in \mathcal{P}$, *let* φ_p^{st} *be a predicate-update formula over vocabulary* $\mathcal{P} \cup \{new\}$. *Then, applying the operational semantics of* $st \equiv label:$ `call f()` *to a structure* S , *yields a structure* S'' *defined as follows: First let* S' *be a structure with a new individual,* u_{new} *not in* U^S, *i.e.*

st	$\varphi_p^{st}(v)$
label : call f()	$\varphi_{stack}^{st}(v) \stackrel{\text{def}}{=} stack(v) \vee new(v)$ $\varphi_{cs_{label}}^{st}(v) \stackrel{\text{def}}{=} cs_{label}(v) \vee new(v)$ $\varphi_{top}^{st}(v) \stackrel{\text{def}}{=} new(v)$ $\varphi_{pr}^{st}(v_1, v_2) \stackrel{\text{def}}{=} pr(v_1, v_2) \vee (new(v_1) \wedge top(v_2))$
return	$\varphi_{stack}^{st}(v) \stackrel{\text{def}}{=} stack(v) \wedge \neg top(v)$ $\varphi_{cs_{label}}^{st}(v) \stackrel{\text{def}}{=} cs_{label}(v) \wedge \neg top(v)$ $\varphi_{top}^{st}(v) \stackrel{\text{def}}{=} \exists v_1 : top(v_1) \wedge pr(v_1, v)$ $\varphi_{pr}^{st}(v_1, v_2) \stackrel{\text{def}}{=} pr(v_1, v_2) \wedge \neg top(v_1)$ $\varphi_{x}^{st}(v_1, v_2) \stackrel{\text{def}}{=} x(v_1, v_2) \wedge \neg top(v_1)$

Table A.2: The predicate-update formulae defining the operational semantics of the call and return statements for the core predicates. `x` is any local variable.

$U^{S'} = U^S \cup \{u_{new}\}$ and a new unary predicate $new(v)$, which holds only for u_{new}, i.e.,

$$p^{S'}(u_1, \ldots, u_k) = \begin{cases} 1 & p = new \text{ and } u_1 = u_{new} \\ 0 & p = new \text{ and } u_1 \neq u_{new} \\ 0 & p \neq new \text{ and there exists } i, \\ & 1 \leq i \leq k, \text{ such that } u_i = u_{new} \\ p^S(u_1, \ldots, u_k) & \text{otherwise} \end{cases}$$

Now define $[\![label: \texttt{call f()}]\!] = S''$ where $U^{S''} = U^{S'}$ and

$$p^{S''}(u_1, \ldots, u_k) = [\![\varphi_p^{st}]\!]_3^{S'}([v_1 \mapsto u_1, \ldots, v_k \mapsto u_k])$$

When a procedure returns, its activation record (which is the current activation record) is removed from the universe. The operational semantics of return is defined in a similar manner to the way a memory disposal statement is defined.

Definition A.3.5 *Let $st \equiv$ `return` For every $p \in \mathcal{P}$, let φ_p^{st} be a predicate-update formula over vocabulary $\mathcal{P}$. Then, applying the operational semantics of $st \equiv$ `return` to a structure S , yields a structure $S' = S''|_{\{u \in U^S : top^S(u)=0\}}$ where $U^{S''} = U^S$ and*

$$p^{S''}(u_1, \ldots, u_k) = [\![\varphi_p^{st}]\!]_3^S([v_1 \mapsto u_1, \ldots, v_k \mapsto u_k])$$

3-valued formulae also provide a natural way to define (conservatively) the meaning of program conditions. In particular, we define the meaning of a condition st to be

$$[\![st]\!](S) \stackrel{\text{def}}{=} [\![\varphi^{st}]\!]_3^S([]).$$

(To keep things simple, we assume that conditions do not have side-effects. It is possible to support side-effects in conditions in the same way that is done for statements, namely, by providing appropriate predicate-update formulae.)

- If $[\![\varphi^{st}]\!]_3^S([])$ yields 1, the condition holds in every store represented by S.
- If $[\![\varphi^{st}]\!]_3^S([])$ yields 0, the condition does not hold in any store represented by S.
- If $[\![\varphi^{st}]\!]_3^S([])$ yields 1/2, then we do not know if the condition always holds, never holds, or sometimes holds and sometimes does not hold in the stores represented by S.

3-valued formulae for four types of conditions involving pointer variables are shown in Table A.3. Other kinds of conditions involving pointer variables would either have other formulae, or would be handled via the formula for `UninterpretedCondition`.

The Embedding Theorem immediately implies that the 3-valued interpretation is conservative with respect to every store that can possibly occur at run-time.

st	φ^{st}
`x == y`	$\forall v : x(v) \Leftrightarrow y(v)$
`x != y`	$\exists v : \neg(x(v) \Leftrightarrow y(v))$
`x == NULL`	$\forall v : \neg x(v)$
`x != NULL`	$\exists v : x(v)$
`UninterpretedCondition`	**1/2**

Table A.3: 3-valued formulae for conditions involving pointer variables.

A.4 Test cases.

```
/* create.c */
#include "list.h"
L create(int s)
{
    L tmp, tl;
    if (s <= 0)
        return NULL;

    tl = create(s-1);
    tmp = (L) malloc(*L);
    tmp->n = tl;
    tmp->d = s;
    return tmp;
}
```

Figure A.1: A recursive procedure which creates a list.

```
/* app.c */
#include "list.h"
L app(L p, L q)
{
    L r;
    if (p == NULL)
        return q;

    r = p;
    while (r->n != NULL)
        r = r->n;
    r->n = q;
    return p;
}
```

Figure A.2: A non recursive function which appends the list pointed to by `q` at the end of the list pointed to by `p`.

```
/* delall.c */
#include "list.h"

L delall(L h)
{
    L t;
    if (h == NULL)
        return;

    t = h->n;
    delall(t);
    free(h);
}
```

Figure A.3: A recursive procedure which frees all the elements of a list.

```
/* insert.c */
#include "list.h"

L insert(L h, int k)
{
    L t;
    if (h == NULL&& h->d < k) {
        t = h->n;
        t = insert(t,k);
        h->n = NULL;
        h->n = t ;
        return h;
    }
    t = malloc(*L);
    t->n = h ;
    t->d = k;
    return t ;
}
```

Figure A.4: A recursive procedure which inserts an element into a sorted linked list.

```
/* delete.c */
#include "list.h"

L delete(L h, int k)
{
    L t;
    if (h == NULL)
        return NULL;

    t = h->n ;
    if ( h->d == k ) {
        h->n = NULL ;
        free (h) ;
        return t ;
    }

    t = delete (t ,k ) ;
    h->n = NULL;
    h->n = t ;
    return h ;
}
```

Figure A.5: A recursive procedure which deletes an element from a linked list.

```
/* search.c */
#include "list.h"

L search(L h, int k)
{
    if (h == NULL)
        return NULL ;

    if ( h->d == k )
        return h ;

    h = h->n ;
    h = search(h ,k);
    return h;
}
```

Figure A.6: A recursive procedure which searches for an element in a list.

```
/* app_r.c */
#include "list.h"

L app_r(L p, L q)
{
    L t ;
    if (p == NULL) {
        return q ;
    }

    t = p->n ;
    t = app_r(t,q);

    p->n = NULL ;
    p->n = t;
    return p ;
}
```

Figure A.7: A recursive procedure which adds one list to the end of another.

```
/* rev_d.c */
#include "list.h"

L rev_d(L r)
{
    L y,t ;

    if (r != NULL) {
    t = r->n;
    t = rev_d(t);
    y = r->n;
    r->n = NULL;

    if (y != NULL) {
        y->n = NULL;
        y->n = r;
    }
    else
        t = r ;

    return t ;
}
```

Figure A.8: A recursive procedure which reverses a list destructively.