8

Pointer Logic

8.1 Introduction

8.1.1 Pointers and Their Applications

This chapter introduces a theory for reasoning about programs that use pointers, and describes decision procedures for it. We assume that the reader is familiar with pointers and their use in programming languages.

A **pointer** is a program variable whose sole purpose is to refer to some other program construct. This other construct could be a variable, a procedure or label, or yet another pointer. Among other things, pointers allow a piece of code to operate on different sets of data, which avoids inefficient copying of data.

As an example, consider a program that maintains two arrays of integers, named `A` and `B`, and that both arrays need to be sorted at some point within the program. Without pointers, the programmer needs to maintain two implementations of the sorting algorithm, one for `A` and one for `B`. Using pointers, a single implementation of sorting is implemented as a procedure that accepts a pointer to the first element of an array as an argument. It is called twice, with the addresses of `A` and `B`, respectively, as the argument.

As pointers are a common source of programming errors, most modern programming languages try to offer alternatives, e.g., in the form of references or abstract data containers. Nevertheless, low-level programming languages with explicit pointers are still frequently used, for example, for embedded systems or operating systems.

The implementation of pointers relies on the fact that the memory cells of a computer have *addresses*, i.e., each cell has a unique number. The value of a pointer is then nothing but such a number. The way the memory cells are addressed is captured by the concept of the **memory model** of the architecture that executes the program.

Definition 8.1 (memory model). *A* memory model *describes the assumptions that are made about the way memory cells are addressed. We assume*

D. Kroening and O. Strichman, *Decision Procedures*,
Texts in Theoretical Computer Science. An EATCS Series,
DOI 10.1007/978-3-662-50497-0_8

$\boxed{M, A}$ $\boxed{D}$ *that the architecture provides a continuous, uniform address space, i.e., the set of addresses A is a subinterval of the integers $\{0, \ldots, N-1\}$. Each address corresponds to a memory cell that is able to store one data word. The set of data words is denoted by D. A* **memory valuation** *$M : A \longrightarrow D$ is a mapping from a set of addresses A into the domain D of data words.*

$\boxed{\sigma}$ A variable may require more than one data word to be stored in memory. For example, this is the case when the variable is of type struct, array, or double-precision floating point. Let $\sigma(v)$ with $v \in V$ denote the size (in data words) of v.

$\boxed{V}$ The compiler assigns a particular memory location (address) to each global, and thus, static variable.[1] This mapping is called the **memory layout**, and is formalized as follows. Let V denote the set of variables.

$\boxed{L}$ **Definition 8.2 (memory layout).** *A* memory layout *$L : V \longrightarrow A$ is a mapping from each variable $v \in V$ to an address $a \in A$. The address of v is also called the* memory location *of v.*

The memory locations of the statically allocated variables are usually assigned such that they are *nonoverlapping* (we explain later on how to model dynamically allocated data structures). Note that the memory layout is not necessarily continuous, i.e., compilers may generate a layout that contains "holes".[2]

Example 8.3. Figure 8.1 illustrates a memory layout for a fragment of an ANSI-C program. The program has six objects, which are named `var_a`, `var_b`, `var_c`, `S`, `array`, and `p`. The first five objects either are integer variables or are composed of integer variables. The object named `p` is a pointer variable, which we assume to be as wide as an integer.[3] The program initializes `p` to the address of the variable `var_c`, which is denoted by `&var_c`. Besides the variable definitions, the program also has a function `main()`, which sets the value of the variable pointed to by `p` to 100. ▟

8.1.2 Dynamic Memory Allocation

Pointers also enable the creation of *dynamic data structures*. Dynamic data structures rely on an area of memory that is designated for use by objects that

[1] Statically allocated variables are variables that are allocated space during the entire run time of the program. In contrast, the addresses of dynamically allocated data such as local variables or data on the heap are determined at run time once the object has been created.

[2] A possible reason for such holes is the need for proper alignment. As an example, many 64-bit architectures are unable to read double-precision floating-point values from addresses that are not a multiple of 8.

[3] This is not always the case; for example, in the x86 16-bit architecture, integers have 16 bits, whereas pointers are 32 bits wide. In some 64-bit architectures, integers have 32 bits, whereas pointers have 64 bits.

```
int var_a, var_b, var_c;
struct { int x; int y; } S;
int array[4];
int *p = &var_c;

int main() {
  *p=100;
}
```

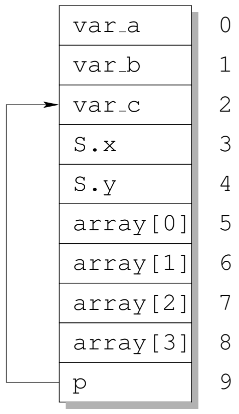


Fig. 8.1. A fragment of an ANSI-C program and a possible memory layout for it

Aside: Pointers and References in Object-Oriented Programming
Separation of data and algorithms is promoted by the concept of *object-oriented programming* (OOP). In modern programming languages such as Java and C++, the explicit use of pointer variables is deprecated. Instead, the procedures that are associated with an object (the *methods*) implicitly receive a pointer to the data members (the *fields*) of the object instance as an argument. In C++, the pointer is accessible using the keyword `this`. All accesses to the data members are performed indirectly by means of the `this` pointer variable.

References, just like pointers, are program variables that refer to a variable or object. The difference between references and pointers is often only syntactic. As an example, the fact that dereferencing is performed is usually hidden. In program analysis, references can be treated just as pointers.

are created at the run time of the program. A run time library maintains a list of the memory regions that are unused. A function, which is part of this library, allocates a region of given size and returns a pointer to the beginning (lowest address) of the region. The memory layout therefore *changes during the run time of the program.* Memory allocation may be performed an unbounded number of times (provided enough space is deallocated as well), and thus, there is no bound on the number of objects that a program can generate.

The function that performs the allocation is called `malloc()` in C, and is provided as an operator called `new` in C++, C#, and Java. In either case, the size of the region that is requested is passed as an argument. In order to reuse memory occupied by data structures that are no longer needed, C programmers call `free`, C++ programmers use `delete`, while Java and C# provide an automatic garbage collection mechanism. The **lifetime** of a dynamic object is the time between its allocation and its deallocation.

8.1.3 Analysis of Programs with Pointers

All but trivial programs rely on pointers or references in order to separate between data and algorithms. Decision procedures that are used for program analysis therefore often need to include reasoning about pointers.

As a simple example, consider the following program fragment, which computes the sum of an array of size 10:

```
void f(int *sum) {
  *sum = 0;

  for(i=0; i<10; i++)
    *sum = *sum + array[i];
}
```

The sum is stored in an integer variable that is pointed to by a pointer called `sum`. Any analysis method that aims at validating the correctness of this fragment has to take the value of the pointer into account. In particular, the program is likely to fail if the address held by `sum` is equal to the address of `i`. In this case, we say that `*sum` is an **alias** for `i`. Aliasing that is not anticipated by the programmer is a common source of problems.

The use of pointers gives rise to program properties that are of high interest. It is well known that many programs fail owing to incorrect use of pointer variables. A very common problem in programs is dereferencing of pointer variables that do not point to a proper object. The value 0 is typically reserved as a designated `NULL` pointer. It is guaranteed that no object, either statically or dynamically allocated, has this address. This value can therefore be used to indicate special cases, for example, the end of a linked list. However, if such a pointer is—by mistake—dereferenced, modern architectures typically generate an exception, which terminates the program.

Programming languages that offer explicit deallocation face another problem. In the following program fragment, an array-type object is allocated and deallocated:

```
int *p, *q;

p = new int[10];
q = &p[3];
delete p;
*q = 2;
```

Note that the address of the fourth element of the array is stored in `q`, and that this pointer is dereferenced after the deallocation of the array. In a variant of the program above, the library that manages the dynamically allocated memory may have reassigned the space used for the array by that time, and thus another object might be overwritten by writing to `*q`. Such errors are hard to reproduce, as they depend on the exact memory layout of the architecture.

They often remain undetected despite extensive testing. The detection of such errors is therefore an important application for static program analysis tools.

Aside: Alias Analysis
Alias analysis has a significant role in pointer-related reasoning about software, such as the analysis performed by optimizing compilers. Alias analysis may be performed at various levels of precision. For example, alias analysis may be field sensitive or insensitive, interprocedural or intraprocedural, and may or may not be sensitive to the control flow. Alias analysis is a special case of *static analysis*, and is typically performed as a *may-analysis*, that is, it determines the set of variables that a given pointer *may* point to — this is called the "points-to" set. In other words, variables that are *not* in this set cannot be pointed to by this pointer. For example, given an instruction such as

```
*p=0;
```

may-analysis permits us to conclude that any variable that is *not* in the points-to set of p is also not modified by this assignment. In the case of an optimizing compiler, this permits us to determine the set of variables that can be cached safely in processor registers.

Alias analysis is performed by maintaining a points-to set for each pointer (and, if desired, for each program location), and updating these sets according to the program statements. The algorithm terminates once the sets have saturated, i.e., do not change anymore.

As an example, consider a control-flow-insensitive analysis of a program with three statements:

```
p=q;
q=&i;
p=&j;
```

The points-to sets of p and q are initially empty. Processing the first statement results in no change. The second statement adds i to the points-to set of q, and the third adds j to the points-to set of p. Owing to the first statement, the set of q is added to that of p and, thereafter, the two sets are saturated.

8.2 A Simple Pointer Logic

8.2.1 Syntax

There are many variants of pointer logic, each with a different syntax and meaning. The more complex ones are often undecidable. We define a simple logic here, with the goal of making the problem of deciding formulas in this logic easier to solve.

Definition 8.4 (pointer logic). *The syntax of a formula in* pointer logic *is defined by the following rules:*

$$\begin{aligned}
\textit{formula} &: \textit{formula} \wedge \textit{formula} \mid \neg\textit{formula} \mid (\textit{formula}) \mid \textit{atom} \\
\textit{atom} &: \textit{pointer} \;=\; \textit{pointer} \mid \textit{term} \;=\; \textit{term} \mid \\
&\quad\; \textit{pointer} \;<\; \textit{pointer} \mid \textit{term} \;<\; \textit{term} \\
\textit{pointer} &: \textit{pointer-identifier} \mid \textit{pointer} + \textit{term} \mid (\textit{pointer}) \mid \\
&\quad\; \&\textit{identifier} \mid \& * \textit{pointer} \mid * \textit{pointer} \mid \text{NULL} \\
\textit{term} &: \textit{identifier} \mid \; * \textit{pointer} \mid \textit{term}\ \textit{op}\ \textit{term} \mid (\textit{term}) \mid \\
&\quad\; \textit{integer-constant} \mid \textit{identifier}\ [\ \textit{term}\ ] \\
\textit{op} &: + \mid -
\end{aligned}$$

The variables represented by *pointer-identifier* are assumed to be of pointer type, whereas the variables represented by *identifier* are assumed to be integers or an array of integers.[4] Note that the grammar allows pointer arithmetic, whereas it prohibits a direct conversion of an integer into a pointer or vice versa. This is motivated by the fact that the conversion of a pointer to an integer may actually fail in a number of architectures, owing to the fact that pointers are wider than the standard integer type.[5]

Example 8.5. Let p, q denote pointer identifiers, and let i, j denote integer identifiers. The following expressions are well formed according to the grammar above:

- $*(p+i) = 1$,
- $*(p+*p) = 0$,
- $p = q \wedge *p = 5$,
- $*****p = 1$,
- $p < q$.

The following expressions are not permitted by the grammar:

- $p+i$,
- $p = i$,
- $*(p+q)$,
- $*1 = 1$,
- $p < i$. ▟

Note that the grammar above encompasses all of integer linear arithmetic (Chap. 5) and also a fragment of array logic (Chap. 7). In practice, a logic for pointers is typically combined with a logic for the program expressions, such as bit-vector arithmetic.

[4] The syntax is clearly inspired by that of ANSI-C. Note, however, that we deviate from the ANSI-C syntax in a few points. As an example, in ANSI-C, an array identifier is synonymous with its address.

[5] Much as in C/C++, an indirect conversion by means of the dereferencing operator is still possible.

8.2.2 Semantics

There are numerous ways to assign a meaning to the expressions defined above. We define the semantics by referring to a specific memory layout L (Definition 8.2) and a specific memory valuation M (Definition 8.1), that is, pointer logic formulas are predicates on M, L pairs. The definition uses a reduction to integer arithmetic and array logic, and thus we treat M and L as array types. We also assume that D (the set of data words) is contained in the set of integers.

Definition 8.6 (semantics of pointer logic). *As before let L denote a memory layout and let M denote a valuation of the memory. Let $\mathcal{L}_P$ denote the set of pointer logic expressions, and let $\mathcal{L}_D$ denote the set of expressions permitted by the logic for the data words. We define a meaning for $e \in \mathcal{L}_P$ using the function $\llbracket\cdot\rrbracket : \mathcal{L}_P \longrightarrow \mathcal{L}_D$. The function $\llbracket e \rrbracket$ is defined recursively as given in Fig. 8.2. The expression $e \in \mathcal{L}_P$ is valid if and only if $\llbracket e \rrbracket$ is valid.*

$\llbracket f_1 \wedge f_2 \rrbracket$	$\doteq$	$\llbracket f_1 \rrbracket \wedge \llbracket f_2 \rrbracket$	
$\llbracket \neg f \rrbracket$	$\doteq$	$\neg \llbracket f \rrbracket$	
$\llbracket p_1 = p_2 \rrbracket$	$\doteq$	$\llbracket p_1 \rrbracket = \llbracket p_2 \rrbracket$	where p_1, p_2 are pointer expressions
$\llbracket p_1 < p_2 \rrbracket$	$\doteq$	$\llbracket p_1 \rrbracket < \llbracket p_2 \rrbracket$	where p_1, p_2 are pointer expressions
$\llbracket t_1 = t_2 \rrbracket$	$\doteq$	$\llbracket t_1 \rrbracket = \llbracket t_2 \rrbracket$	where t_1, t_2 are terms
$\llbracket t_1 < t_2 \rrbracket$	$\doteq$	$\llbracket t_1 \rrbracket < \llbracket t_2 \rrbracket$	where t_1, t_2 are terms
$\llbracket p \rrbracket$	$\doteq$	$M[L[p]]$	where p is a pointer identifier
$\llbracket p + t \rrbracket$	$\doteq$	$\llbracket p \rrbracket + \llbracket t \rrbracket$	where p is a pointer expression, and t is a term
$\llbracket \& v \rrbracket$	$\doteq$	$L[v]$	where $v \in V$ is a variable
$\llbracket \& * p \rrbracket$	$\doteq$	$\llbracket p \rrbracket$	where p is a pointer expression
$\llbracket \mathrm{NULL} \rrbracket$	$\doteq$	0	
$\llbracket v \rrbracket$	$\doteq$	$M[L[v]]$	where $v \in V$ is a variable
$\llbracket * p \rrbracket$	$\doteq$	$M[\llbracket p \rrbracket]$	where p is a pointer expression
$\llbracket t_1 \ op\ t_2 \rrbracket$	$\doteq$	$\llbracket t_1 \rrbracket \ op\ \llbracket t_2 \rrbracket$	where t_1, t_2 are terms
$\llbracket c \rrbracket$	$\doteq$	c	where c is an integer constant
$\llbracket v[t] \rrbracket$	$\doteq$	$M[L[v] + \llbracket t \rrbracket]$	where v is an array identifier, and t is a term

Fig. 8.2. Semantics of pointer expressions

Observe that a pointer p points to a variable x if $M[L[p]] = L[x]$, that is, the value of p is equal to the address of x. As a shorthand, we write $p \hookrightarrow z$ to mean that p points to some memory cell such that $*p = z$. Observe also that the meaning of pointer arithmetic, for example, $p + i$, does not depend on the type of object that p points to.[6] $\boxed{p \hookrightarrow z}$

[6] In contrast, the semantics of ANSI-C requires that an integer that is added to a pointer p is multiplied by the size of the type that p points to.

Example 8.7. Consider the following expression, where a is an array identifier:

$$*\,((\&a) + 1) \;=\; a[1]\;. \tag{8.1}$$

The semantic definition of (8.1) expands as follows:

$$\begin{aligned}
[\![*((\&a)+1) \;=\; a[1]]\!] &\iff [\![*((\&a)+1)]\!] \;=\; [\![a[1]]\!] && (8.2)\\
&\iff M[[\![(\&a)+1]\!]] \;=\; M[L[a] + [\![1]\!]] && (8.3)\\
&\iff M[[\![\&a]\!] + [\![1]\!]] \;=\; M[L[a]+1] && (8.4)\\
&\iff M[L[a]+1] \;=\; M[L[a]+1] && (8.5)
\end{aligned}$$

Equation (8.5) is obviously valid, and thus, so is (8.1). Note that the translated formula must evaluate to TRUE for any L and M and, thus, the following formula is not valid:

$$*\,p = 1 \implies x = 1\;. \tag{8.6}$$

For $p \neq \&x$, this formula evaluates to FALSE. ▟

8.2.3 Axiomatization of the Memory Model

Formulas in pointer logic may exploit assumptions made about the memory model. The set of these assumptions depends highly on the architecture. Here, we formalize properties that most architectures comply with, and thus that many programs rely on.

On most architectures, the following two formulas are valid, and hence can be safely assumed by programmers:

$$\&x \neq \text{NULL}\;, \tag{8.7}$$

$$\&x \neq \&y\;. \tag{8.8}$$

Equation (8.7) translates into $L[x] \neq 0$ and relies on the fact that no object has address 0. Equation (8.8) relies on the fact that the memory layout assigns nonoverlapping addresses to the objects. We define a series of *memory model axioms* in order to formalize these properties.

Memory Model Axiom 1 ("No object has address 0") *The fact "no object has address 0" is easily formalized:*[7]

$$\forall v \in V.\; L[v] \neq 0\;. \tag{8.9}$$

[7] Note that the ANSI-C standard does not actually guarantee that the symbolic constant NULL is represented by a bit vector consisting of zeros; however, it guarantees that the NULL pointer compares to the integer zero and can be obtained by converting the integer zero to a pointer type.

The easiest way to ensure that (8.8) is valid is to assume that $\forall v_1, v_2 \in V.\ v_1 \neq v_2 \implies L[v_1] \neq L[v_2]$. However, this assumption is often not strong enough, as objects with size greater or equal to two may still overlap. We therefore assume the following two conditions, which together are stronger:

Memory Model Axiom 2 ("Objects have size at least one") *The fact "an object has size at least one" is easily captured by*

$$\forall v \in V.\ \sigma(v) \geq 1\ . \tag{8.10}$$

Memory Model Axiom 3 ("Objects do not overlap") *Different objects do not share any addresses:*

$$\begin{aligned}\forall v_1, v_2 \in V.\ v_1 \neq v_2 \implies & \{L[v_1], \ldots, L[v_1] + \sigma(v_1) - 1\} \cap \\ & \{L[v_2], \ldots, L[v_2] + \sigma(v_2) - 1\} = \emptyset\ .\end{aligned} \tag{8.11}$$

Program analysis tools that are applied to code that relies on additional, architecture-specific guarantees may require a larger set of memory model axioms. Examples are *byte ordering* and *endianness*, and specific assumptions about *alignment*.

8.2.4 Adding Structure Types

Structure types are a convenient way to implement data structures. Structure types can be added to our pointer logic as a purely syntactic extension, as we shall soon see. We assume that the fields of the structure types are named, and write $s.f$ to denote the value of the field f in the structure s.

Formally, we can view structure types as "syntactic sugar" for array types, and record the following shorthands. Each field of the structure is assigned a unique **offset**. Let $o(f)$ denote the offset of field f. We then define the meaning of $s.f$ as follows: $\boxed{o(f)}$ $\boxed{s.f}$

$$s.f \quad \doteq \quad *((\&s) + o(f))\ . \tag{8.12}$$

For convenience, we introduce two additional shorthands. Following the PASCAL and ANSI-C syntax, we write $p\text{->}f$ for $(*p).f$ (this shorthand is not to be confused with logical implication or with $p \hookrightarrow a$). Adopting some notation from separation logic (see the aside on separation logic on p. 184), we also extend the $p \hookrightarrow a$ notation by introducing $p \hookrightarrow a, b, c, \ldots$ as a shorthand for $\boxed{p\text{->}f}$

$$\begin{aligned} &*(p + 0) = a\ \land \\ &*(p + 1) = b\ \land \\ &*(p + 2) = c\ \ldots\ . \end{aligned} \tag{8.13}$$

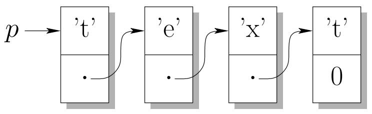
8.3 Modeling Heap-Allocated Data Structures

8.3.1 Lists

Heap-allocated data structures play an important role in programs, and are prone to pointer-related errors. We now illustrate how to model a number of commonly used data structures using pointer logic.

After the array, the simplest dynamically allocated data structure is the *linked list*. It is typically realized by means of a structure type that contains fields for a *next pointer* and the data that are to be stored in the list.

As an example, consider the following list: The first field is named a and is an ASCII character, serving as the "payload", and the second field is named n, and is the pointer to the next element of the list. Following ANSI-C syntax, we use 'x' to denote the integer that represents the ASCII character "x":



The list is terminated by a NULL pointer, which is denoted by "0" in the diagram above. A way of modeling this list is to use the following formula:

$$\begin{array}{rl} & p \hookrightarrow \text{'t'}, p_1 \\ \wedge & p_1 \hookrightarrow \text{'e'}, p_2 \\ \wedge & p_2 \hookrightarrow \text{'x'}, p_3 \\ \wedge & p_3 \hookrightarrow \text{'t'}, \text{NULL} . \end{array} \tag{8.14}$$

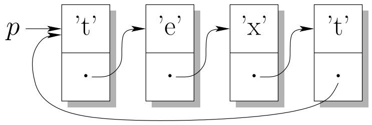
This way of specifying lists is cumbersome, however. Therefore, disregarding the payload field, we first introduce a recursive shorthand for the i-th member of a list:[8]

$$\begin{array}{rl} \textsf{list-elem}(p, 0) \doteq & p , \\ \textsf{list-elem}(p, i) \doteq & \textsf{list-elem}(p, i-1)\text{->}n \quad \text{for } i \geq 1 . \end{array} \tag{8.15}$$

$\boxed{\textsf{list}}$ We now define the shorthand $\textsf{list}(p, l)$ to denote a predicate that is true if p points to a NULL-terminated acyclic list of length l:

$$\textsf{list}(p, l) \doteq \textsf{list-elem}(p, l) = \text{NULL} . \tag{8.16}$$

A linked list is *cyclic* if the pointer of the last element points to the first one:

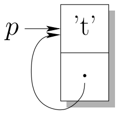


Consider the following variant $\textsf{my-list}(p, l)$, intended to capture the fact that p points to such a cyclic list of length $l \geq 1$:

[8] Note that recursive definitions of this form are, in general, only embeddable into our pointer logic if the second argument is a constant.

$$\textsf{my-list}(p, l) \doteq \textsf{list-elem}(p, l) = p \;. \tag{8.17}$$

Does this definition capture the concept properly? The list in the diagram above satisfies $\textsf{my-list}(p, 4)$. Unfortunately, the following list satisfies $\textsf{my-list}(p, 4)$ just as well:



This is due to the fact that our definition does not preclude *sharing* of elements of the list, despite the fact that we had certainly intended to specify that there are l disjoint list elements. Properties of this kind are often referred to as *separation properties*. A way to assert that the list elements are disjoint is to define a shorthand overlap as follows:

$$\textsf{overlap}(p, q) \doteq \; p = q \,\vee\, p + 1 = q \,\vee\, p = q + 1 \;. \tag{8.18}$$

This shorthand is then used to state that all list elements are pairwise disjoint:

$$\begin{aligned} \textsf{list-disjoint}(p, 0) &\doteq \;\text{TRUE} \;, \\ \textsf{list-disjoint}(p, l) &\doteq \;\textsf{list-disjoint}(p, l-1) \wedge \\ \forall 0 \le i < l - 1.\ &\neg\textsf{overlap}(\textsf{list-elem}(p, i), \textsf{list-elem}(p, l-1)) \;. \end{aligned} \tag{8.19}$$

Note that the size of this formula grows quadratically in l. As separation properties are frequently needed, more concise notations have been developed for this concept, for example, *separation logic* (see the aside on that topic). Separation logic can express such properties with formulas of linear size.

8.3.2 Trees

We can implement a *binary tree* by adding another pointer field to each element of the data structure (see Fig. 8.3). We denote the pointer to the left-hand child node by l, and the pointer to the right-hand child by r.

In order to illustrate a pointer logic formula for trees, consider the tree in Fig. 8.3, which has one integer x as payload. Observe that the integers are arranged in a particular fashion: the integer of the left-hand child of any node n is always smaller than the integer of the node n itself, whereas the integer of the right-hand child of node n is always larger than the integer of the node n. This property permits lookup of elements with a given integer value in time $O(h)$, where h is the height of the tree. The property can be formalized as follows:

$$\begin{aligned} &(n.l \neq \text{NULL} \implies n.l\text{->}x < n.x) \\ \wedge\; &(n.r \neq \text{NULL} \implies n.r\text{->}x > n.x) \;. \end{aligned} \tag{8.22}$$

Unfortunately, (8.22) is not strong enough to imply lookup in time $O(h)$. For this, we need to establish the ordering over the integers of an *entire subtree*.

Aside: Separation Logic
Theories for dynamic data structures are frequently used for proving that memory cells *do not alias*. While it is possible to model the statement that a given object does not alias with other objects with pairwise comparison, reasoning about such formulation scales poorly. It requires enumeration of all heap-allocated objects, which makes it difficult to reason about a program in a local manner.

John Reynolds' *separation logic* [242] addresses both problems by introducing a new binary operator "$*$", as in "$P * Q$", which is called a *separating conjunction*. The meaning of $*$ is similar to the standard Boolean conjunction, i.e., $P \wedge Q$, but it also asserts that P and Q reason about separate, nonoverlapping portions of the heap. As an example, consider the following variant of the list predicate:

$$\begin{aligned} \mathsf{list}(p, 0) &\doteq\ p = \mathrm{NULL} \\ \mathsf{list}(p, l) &\doteq\ \exists q.\ p \hookrightarrow z, q \wedge \mathsf{list}(q, l-1) \quad \text{for } l \geq 1\ . \end{aligned} \tag{8.20}$$

Like our previous definition, the definition above suffers from the fact that some memory cells of the elements of the list might overlap. This can be mended by replacing the standard conjunction in the definition above by a separating conjunction:

$$\mathsf{list}(p, l) \doteq\ \exists q.\ p \hookrightarrow z, q * \mathsf{list}(q, l-1)\ . \tag{8.21}$$

This new list predicate also asserts that the memory cells of all list elements are pairwise disjoint. Separation logic, in its generic form, is not decidable, but a variety of decidable fragments have been identified.

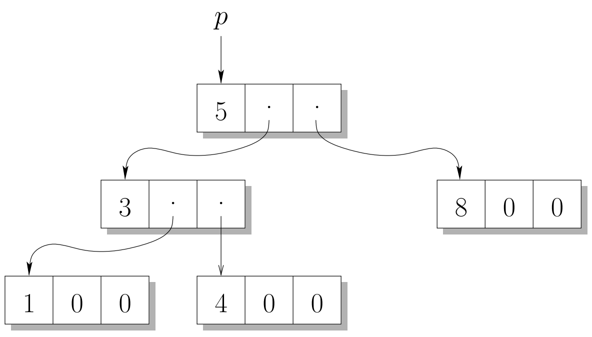


Fig. 8.3. A binary tree that represents a set of integers

We define a predicate $\mathsf{tree\text{-}reach}(p, q)$, which holds if q is reachable from p in one step:

$$\begin{aligned}\mathsf{tree\text{-}reach}(p,q) \doteq\ & p \neq \text{NULL} \wedge q \neq \text{NULL} \wedge \\ & (p = q \vee p\text{->}l = q \vee p\text{->}r = q)\ .\end{aligned} \tag{8.23}$$

In order to obtain a predicate that holds if and only if q is reachable from p in any number of steps, we define the *transitive closure* of a given binary relation R.

Definition 8.8 (transitive closure). *Given a binary relation R, the* transitive closure TC_R *relates x and y if there are $z_1, z_2, \ldots, z_n$ such that*

$$xRz_1 \wedge z_1Rz_2 \wedge \ldots \wedge z_nRy\ .$$

Formally, transitive closure can be defined inductively as follows:

$$\begin{aligned}\mathsf{TC}^1_R(p,q) &\doteq R(p,q)\ ,\\ \mathsf{TC}^i_R(p,q) &\doteq \exists p'.\ \mathsf{TC}^{i-1}_R(p,p') \wedge R(p',q)\ ,\\ \mathsf{TC}(p,q) &\doteq \exists i.\ \mathsf{TC}^i_R(p,q)\ .\end{aligned} \tag{8.24}$$

Using the transitive closure of our $\mathsf{tree\text{-}reach}$ relation, we obtain a new relation $\mathsf{tree\text{-}reach^*}(p, q)$ that holds if and only if q is reachable from p in any number of steps:

$$\mathsf{tree\text{-}reach^*}(p,q) \iff \mathsf{TC}_{\mathsf{tree\text{-}reach}}(p,q)\ . \tag{8.25}$$

Using $\mathsf{tree\text{-}reach^*}$, it is easy to strengthen (8.22) appropriately:

$$\begin{aligned}& (\forall p.\ \mathsf{tree\text{-}reach^*}(n.l,p) \implies p\text{->}x < n.x)\\ \wedge\ & (\forall p.\ \mathsf{tree\text{-}reach^*}(n.r,p) \implies p\text{->}x > n.x)\ .\end{aligned} \tag{8.26}$$

Unfortunately, the addition of the transitive closure operator can make even simple logics undecidable, and thus, while convenient for modeling, it is a burden for automated reasoning. We restrict the presentation below to decidable cases by considering only special cases.

8.4 A Decision Procedure

8.4.1 Applying the Semantic Translation

The semantic translation introduced in Sect. 8.2.2 not only assigns meaning to the pointer formulas, but also gives rise to a simple decision procedure. The formulas generated by this semantic translation contain array read operators and linear arithmetic over the type that is used for the indices. This may be the set of integers (Chap. 5) or the set of bit vectors (Chap. 6). It also contains at least equalities over the type that is used to model the contents of the memory cells. We assume that this is the same type as the index type. As we have seen in Chap. 7, such a logic is decidable. Care has to be taken when extending the pointer logic with quantification, as array logic with arbitrary quantification is undecidable.

A straightforward decision procedure for pointer logic therefore first applies the semantic translation to a pointer formula φ to obtain a formula φ' in the combined logic of linear arithmetic over integers and arrays of integers. The formula φ' is then passed to the decision procedure for the combined logic. As the formulas φ and φ' are equisatisfiable (by definition), the result returned for φ' is also the correct result for φ.

Example 8.9. Consider the following pointer logic formula, where x is a variable, and p identifies a pointer:

$$p = \&x \wedge x = 1 \implies *p = 1 \ . \tag{8.27}$$

The semantic definition of this formula expands as follows:

$$\begin{aligned} &[\![p = \&x \wedge x = 1 \implies *p = 1]\!] \\ &\quad\iff [\![p = \&x]\!] \wedge [\![x = 1]\!] \implies [\![*p = 1]\!] \\ &\quad\iff [\![p]\!] = [\![\&x]\!] \wedge [\![x]\!] = 1 \implies [\![*p]\!] = 1 \\ &\quad\iff M[L[p]] = L[x] \wedge M[L[x]] = 1 \implies M[M[L[p]]] = 1 \ . \end{aligned} \tag{8.28}$$

A decision procedure for array logic and equality logic easily concludes that the formula above is valid, and thus, so is (8.27).

As an example of an invalid formula, consider

$$p \hookrightarrow x \implies p = \&x \ . \tag{8.29}$$

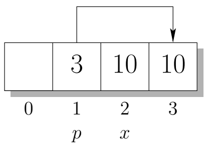
The semantic definition of this formula expands as follows:

$$\begin{aligned} &[\![p \hookrightarrow x \implies p = \&x]\!] \\ &\quad\iff [\![p \hookrightarrow x]\!] \implies [\![p = \&x]\!] \\ &\quad\iff [\![*p = x]\!] \implies [\![p]\!] = [\![\&x]\!] \\ &\quad\iff [\![*p]\!] = [\![x]\!] \implies M[L[p]] = L[x] \\ &\quad\iff M[M[L[p]]] = M[L[x]] \implies M[L[p]] = L[x] \end{aligned} \tag{8.30}$$

A counterexample to this formula is the following:

$$L[p] = 1,\ L[x] = 2,\ M[1] = 3,\ M[2] = 10,\ M[3] = 10 \ . \tag{8.31}$$

The values of M and L in the counterexample are best illustrated with a picture:



∎

Applying the Memory Model Axioms

A formula may rely on one of the memory model axioms defined in Sect. 8.2.3. As an example, consider the following formula:

$$\sigma(x) = 2 \implies \&y \neq \&x + 1 \; . \tag{8.32}$$

The semantic translation yields

$$\sigma(x) = 2 \implies L[y] \neq L[x] + 1 \; . \tag{8.33}$$

This formula can be shown to be valid by instantiating Memory Model Axiom 3. After instantiating v_1 with x and v_2 with y, we obtain

$$\{L[x], \ldots, L[x] + \sigma(x) - 1\} \cap \{L[y], \ldots, L[y] + \sigma(y) - 1\} = \emptyset \; . \tag{8.34}$$

We can transform the set expressions in (8.34) into linear arithmetic over the integers as follows:

$$(L[x] + \sigma(x) - 1 < L[y]) \lor (L[x] > L[y] + \sigma(y) - 1) \; . \tag{8.35}$$

Using $\sigma(x) = 2$ and $\sigma(y) \geq 1$ (Memory Model Axiom 2), we can conclude, furthermore, that

$$(L[x] + 1 < L[y]) \lor (L[x] > L[y]) \; . \tag{8.36}$$

Equation (8.36) is strong enough to imply $L[y] \neq L[x] + 1$, which proves that Eq. (8.32) is valid.

8.4.2 Pure Variables

The semantic translation of a pointer formula results in a formula that we can decide using the procedures described in this book. However, semantic translation down to memory valuations places an undue burden on the underlying decision procedure, as illustrated by the following example (symmetry of equality):

$$\begin{aligned} & [\![x = y \implies y = x]\!] && (8.37)\\ \iff & [\![x = y]\!] \implies [\![y = x]\!] && (8.38)\\ \iff & M[L[x]] = M[L[y]] \implies M[L[y]] = M[L[x]] \; . && (8.39) \end{aligned}$$

A decision procedure for array logic and equality logic is certainly able to deduce that (8.39) is valid. Nevertheless, the steps required for solving (8.39) obviously exceed the effort required to decide

$$x = y \implies y = x \; . \tag{8.40}$$

In particular, the semantic translation does not exploit the fact that x and y do not actually interact with any pointers. A straightforward optimization is therefore the following: if the address of a variable x is not referred to, we translate it to a new variable Υ_x instead of $M[L[x]]$. A formalization of this idea requires the following definition:

Definition 8.10 (pure variables). *Given a formula φ with a set of variables V, let $\mathcal{P}(\varphi) \subseteq V$ denote the subset of φ's variables that are not used within an argument of the "&" operator within φ. These variables are called* pure.

As an example, $\mathcal{P}(\&x = y)$ is $\{y\}$. We now define a new translation function $[\![\cdot]\!]^{\mathcal{P}}$. The definition of $[\![e]\!]^{\mathcal{P}}$ is identical to the definition of $[\![e]\!]$ unless e denotes a variable in $\mathcal{P}(\varphi)$. The new definition is:

$$\begin{array}{rcll} [\![v]\!]^{\mathcal{P}} & \doteq & \Upsilon_v & \text{for } v \in \mathcal{P}(\varphi) \\ [\![v]\!]^{\mathcal{P}} & \doteq & M[L[v]] & \text{for } v \in V \setminus \mathcal{P}(\varphi) \end{array}$$

Theorem 8.11. *The translation using pure variables is equisatisfiable with the semantic translation:*

$$[\![\varphi]\!]^{\mathcal{P}} \iff [\![\varphi]\!] \,.$$

Example 8.12. Equation (8.38) is now translated as follows without referring to a memory valuation, and thus no longer burdens the decision procedure for array logic:

$$[\![x = y \implies y = x]\!]^{\mathcal{P}} \tag{8.41}$$

$$\iff [\![x = y \implies y = x]\!]^{\mathcal{P}} \tag{8.42}$$

$$\iff [\![x = y]\!]^{\mathcal{P}} \implies [\![y = x]\!]^{\mathcal{P}} \tag{8.43}$$

$$\iff \Upsilon_x = \Upsilon_y \implies \Upsilon_y = \Upsilon_x \,. \tag{8.44}$$

▟

8.4.3 Partitioning the Memory

The translation procedure can be optimized further using the following observation: the run time of a decision procedure for array logic depends on the number of different expressions that are used to index a particular array (see Chap. 7). As an example, consider the pointer logic formula

$$*\,p = 1 \wedge *q = 1 \,, \tag{8.45}$$

which—using our optimized translation—is reduced to

$$M[\Upsilon_p] = 1 \wedge M[\Upsilon_q] = 1 \,. \tag{8.46}$$

The pointers p and q might alias, but there is no reason why they have to. Without loss of generality, we can therefore safely assume that they do not alias and, thus, we partition M into M_1 and M_2:

$$M_1[\Upsilon_p] = 1 \wedge M_2[\Upsilon_q] = 1 \,. \tag{8.47}$$

While this has increased the number of array variables, the number of different indices *per array* has decreased. Typically, this improves the performance of a decision procedure for array logic.

This transformation cannot always be applied, illustrated by the following example:

$$p = q \implies *p = *q \; . \tag{8.48}$$

This formula is obviously valid, but if we partition as before, the translated formula is no longer valid:

$$\Upsilon_p = \Upsilon_q \implies M_1[\Upsilon_p] = M_2[\Upsilon_q] \; . \tag{8.49}$$

Unfortunately, deciding if the optimization is applicable is in general as hard as deciding φ itself. We therefore settle for an approximation based on a syntactic test. This approximation is conservative, i.e., sound, while it may not result in the best partitioning that is possible in theory.

Definition 8.13. *We say that two pointer expressions p and q are* related *directly by a formula φ if both p and q are used inside the same relational expression in φ and that the expressions are related transitively if there is a pointer expression p' that relates to p and relates to q. We write $p \approx q$ if p and q are related directly or transitively.* $\boxed{p \approx q}$

The relation $\approx$ induces a partitioning of the pointer expressions in φ. We number these partitions $1, \ldots, n$. Let $I(p) \in \{1, \ldots, n\}$ denote the index of the partition that p is in. We now define a new translation $[\![\cdot]\!]^{\approx}$, in which we use a separate memory valuation $M_{I(p)}$ when p is dereferenced. The definition of $[\![e]\!]^{\approx}$ is identical to the definition of $[\![e]\!]^{\mathcal{P}}$ unless e is a dereferencing expression. In this case, we use the following definition:

$$[\![*p]\!]^{\approx} \doteq M_{I(p)}([\![p]\!]^{\approx}) \; .$$

Theorem 8.14. *Translation using memory partitioning results in a formula that is equisatisfiable with the result of the semantic translation:*

$$\exists \alpha_1 . \, \alpha_1 \models [\![\varphi]\!]^{\approx} \quad \iff \quad \exists \alpha_2 . \, \alpha_2 \models [\![\varphi]\!] \; .$$

Note that the theorem relies on the fact that our grammar does not permit explicit restrictions on the memory layout L. The theorem no longer holds as soon as this restriction is lifted (see Problem 8.5).

8.5 Rule-Based Decision Procedures

With pointer logics expressive enough to model interesting data structures, one often settles for incomplete, rule-based procedures. The basic idea of such procedures is to define a fragment of pointer logic enriched with predicates for specific types of data structures (e.g., lists or trees) together with a set of proof rules that are sufficient to prove a wide range of verification conditions that arise in practice. The soundness of these proof rules is usually shown with respect to the definitions of the predicates, which implies soundness of the decision procedure. There are only a few known proof systems that are provably complete.

8.5.1 A Reachability Predicate for Linked Structures

As a simple example of this approach, we present a variant of a calculus for reachability predicates introduced by Greg Nelson [204]. Further rule-based reasoning systems are discussed in the bibliographic notes at the end of this chapter.

We first generalize the $\mathsf{list\text{-}elem}$ shorthand used before for specifying linked lists by parameterizing it with the name of the field that holds the pointer to the "next" element. Suppose that f is a field of a structure and holds a pointer. The shorthand $\mathsf{follow}^f_n(q)$ stands for the pointer that is obtained by starting from q and following the field f, n times:

$$\begin{aligned}\mathsf{follow}^f_0(p) &\doteq p\\ \mathsf{follow}^f_n(p) &\doteq \mathsf{follow}^f_{n-1}(p)\text{->}f\ .\end{aligned} \tag{8.50}$$

If $\mathsf{follow}^f_n(p) = q$ holds, then q is reachable in n steps from p by following f. We say that q is reachable from p by following f if there exists such n. Using this shorthand, we enrich the logic with just a single predicate for list-like data structures, denoted by

$$p \overset{f}{\underset{x}{\to}} q\ , \tag{8.51}$$

which is called a **reachability predicate**. It is read as "q is reachable from p following f, while avoiding x". It holds if two conditions are fulfilled:

1. There exists some n such that q is reachable from p by following f n times.
2. x is not reachable in fewer than n steps from p following f.

This can be formalized using $\mathsf{follow}()$ as follows:

$$p \overset{f}{\underset{x}{\to}} q \iff \exists n.(\mathsf{follow}^f_n(p) = q \wedge \forall m < n.\mathsf{follow}^f_m(p) \neq x)\ . \tag{8.52}$$

We say that a formula is a **reachability predicate formula** if it contains the reachability predicate.

Example 8.15. Consider the following software verification problem. The following program fragment iterates over an acyclic list and searches for a list entry with payload a:

```
struct S { struct S *nxt; int payload; } *list;

...
bool find(int a) {
  for(struct S *p=list; p!=0; p=p->nxt)
    if(p->payload==a) return true;
  return false;
}
```

We can specify the correctness of the result returned by this procedure using the following formula:

$$\texttt{find}(a) \iff \exists p'.(\mathit{list} \overset{nxt}{\underset{0}{\to}} p' \land p'\text{->}\mathit{payload} = a)\ . \tag{8.53}$$

Thus, $\texttt{find}(a)$ is true if the following conditions hold:

1. There is a list element that is reachable from *list* by following *nxt* without passing through a NULL pointer.
2. The payload of this list element is equal to a.

We annotate the beginning of the loop body in the program above with the following loop invariant, denoted by INV:

$$\mathsf{INV} := \ \mathit{list} \overset{nxt}{\underset{0}{\to}} p \land (\forall q \neq p.\ \mathit{list} \overset{nxt}{\underset{p}{\to}} q \implies q\text{->}\mathit{payload} \neq a)\ . \tag{8.54}$$

Informally, we make the following argument: first, we show that the program maintains the loop invariant INV; then, we show that INV implies our property.

Formally, this is shown by means of four **verification conditions**. The validity of all of these verification conditions implies the property. We use the notation $e[x/y]$ to denote the expression e in which x is replaced by y.

$$\mathsf{IND\text{-}BASE} := p = \mathit{list} \implies \mathsf{INV} \tag{8.55}$$

$$\mathsf{IND\text{-}STEP} := (\mathsf{INV} \land p\text{->}\mathit{payload} \neq a) \implies \mathsf{INV}[p/p\text{->}\mathit{nxt}] \tag{8.56}$$

$$\begin{aligned}\mathsf{VC\text{-}P1} := &\ (\mathsf{INV} \land p\text{->}\mathit{payload} = a) \\ &\implies \exists p'.(\mathit{list} \overset{nxt}{\underset{0}{\to}} p' \land p'\text{->}\mathit{payload} = a)\end{aligned} \tag{8.57}$$

$$\mathsf{VC\text{-}P2} := (\mathsf{INV} \land p = 0) \implies \neg\exists p'.(\mathit{list} \overset{nxt}{\underset{0}{\to}} p' \land p'\text{->}\mathit{payload} = a) \tag{8.58}$$

The first verification condition, IND-BASE, corresponds to the induction base of the inductive proof. It states that INV holds upon entering the loop, because at that point $p = \mathit{list}$. The formula IND-STEP corresponds to the induction step: it states that the loop invariant is maintained if another loop iteration is executed (i.e., $p\text{->}\mathit{payload} \neq a$).

The formulas VC-P1 and VC-P2 correspond to the two cases of leaving the find function: VC-P1 establishes the property if TRUE is returned, and VC-P2 establishes the property if FALSE is returned. Proving these verification conditions therefore shows that the program satisfies the required property. ▟

8.5.2 Deciding Reachability Predicate Formulas

As before, we can simply expand the definition above and obtain a semantic reduction. As an example, consider the verification condition labeled IND-BASE in Sect. 8.5.1:

$$p = list \implies \mathsf{INV} \tag{8.59}$$

$$\iff p = list \implies list \underset{0}{\overset{nxt}{\rightarrow}} p \wedge \forall q \neq p.\ list \underset{p}{\overset{nxt}{\rightarrow}} q \implies q\text{->}payload \neq a \tag{8.60}$$

$$\iff list \underset{0}{\overset{nxt}{\rightarrow}} list \wedge \forall q \neq list.\ (list \underset{list}{\overset{nxt}{\rightarrow}} q \implies q\text{->}payload \neq a) \tag{8.61}$$

$$\begin{aligned}\iff &(\exists n.\ \mathsf{follow}_n^{nxt}(list) = list \wedge \forall m < n.\ \mathsf{follow}_m^{nxt}(list) \neq list) \wedge \\ &(\forall q \neq list.\ ((\exists n.\ \mathsf{follow}_n^{nxt}(list) = q \wedge \forall m < n.\ \mathsf{follow}_m^{nxt}(list) \neq list) \\ &\quad \implies q\text{->}payload \neq a))\ .\end{aligned} \tag{8.62}$$

Equation (8.62) is argued to be valid as follows: In the first conjunction, instantiate n with 0. In the second conjunct, observe that $q \neq list$, and thus any n satisfying $\exists n.\ \mathsf{follow}_n^{nxt}(list) = q$ must be greater than 0. Finally, observe that $\mathsf{follow}_m^{nxt}(list) \neq list$ is invalid for $m = 0$, and thus the left-hand side of the implication is FALSE.

However, note that the formulas above contain many existential and universal quantifiers over natural numbers and pointers. Applying the semantic reduction therefore does not result in a formula that is in the array property fragment defined in Chap. 7. Thus, the decidability result shown in that chapter does not apply here. How can such complex reachability predicate formulas be solved?

Using Rules

In such situations, the following technique is frequently applied: *rules* are derived from the semantic definition of the predicate, and then they are applied to simplify the formula.

$$p \underset{x}{\overset{f}{\rightarrow}} q \iff (p = q \vee (p \neq x \wedge p\text{->}f \underset{x}{\overset{f}{\rightarrow}} q)) \tag{A1}$$

$$(p \underset{x}{\overset{f}{\rightarrow}} q \wedge q \underset{x}{\overset{f}{\rightarrow}} r) \implies p \underset{x}{\overset{f}{\rightarrow}} r \tag{A2}$$

$$p \underset{x}{\overset{f}{\rightarrow}} q \implies p \underset{q}{\overset{f}{\rightarrow}} q \tag{A3}$$

$$(p \underset{y}{\overset{f}{\rightarrow}} x \wedge p \underset{z}{\overset{f}{\rightarrow}} y) \implies p \underset{z}{\overset{f}{\rightarrow}} x \tag{A4}$$

$$(p \underset{x}{\overset{f}{\rightarrow}} x \vee p \underset{y}{\overset{f}{\rightarrow}} y) \implies (p \underset{y}{\overset{f}{\rightarrow}} x \vee p \underset{x}{\overset{f}{\rightarrow}} y) \tag{A5}$$

$$(p \underset{y}{\overset{f}{\rightarrow}} x \wedge p \underset{z}{\overset{f}{\rightarrow}} y) \implies x \underset{z}{\overset{f}{\rightarrow}} y \tag{A6}$$

$$p\text{->}f \underset{q}{\overset{f}{\rightarrow}} q \iff p\text{->}f \underset{p}{\overset{f}{\rightarrow}} q \tag{A7}$$

Fig. 8.4. Rules for the reachability predicate

The rules provided in [204] for our reachability predicate are given in Fig. 8.4. The first rule ($A1$) corresponds to a program fragment that follows field f once. If q is reachable from p, avoiding x, then either $p = q$ (we are already there) or $p \neq x$, and we can follow f from p to get to a node from which q is reachable, avoiding x. We now prove the correctness of this rule.

Proof. We first expand the definition of our reachability predicate:

$$p \xrightarrow[x]{f} q \iff \exists n.\ (\mathsf{follow}_n^f(p) = q \wedge \forall m < n.\ \mathsf{follow}_m^f(p) \neq x)\ . \tag{8.63}$$

Observe that for any natural n, $n = 0 \vee n > 0$ holds, which we can therefore add as a conjunct:

$$\begin{aligned} \iff \exists n.\ ((n = 0 \vee n > 0) \wedge \\ \mathsf{follow}_n^f(p) = q \wedge \forall m < n.\ \mathsf{follow}_m^f(p) \neq x)\ . \end{aligned} \tag{8.64}$$

This simplifies as follows:

$$\iff \exists n.\ p = q \vee (n > 0 \wedge \mathsf{follow}_n^f(p) = q \wedge \forall m < n.\ \mathsf{follow}_m^f(p) \neq x) \tag{8.65}$$

$$\iff p = q \vee \exists n > 0.\ (\mathsf{follow}_n^f(p) = q \wedge \forall m < n.\ \mathsf{follow}_m^f(p) \neq x)\ . \tag{8.66}$$

We replace n by $n' + 1$ for natural n':

$$\iff p = q \vee \exists n'.\ (\mathsf{follow}_{n'+1}^f(p) = q \wedge \forall m < n' + 1.\ \mathsf{follow}_m^f(p) \neq x)\ . \tag{8.67}$$

As $\mathsf{follow}_{n'+1}^f(p) = \mathsf{follow}_{n'}^f(p\text{->}f)$, this simplifies to

$$\iff p = q \vee \exists n'.\ (\mathsf{follow}_{n'}^f(p\text{->}f) = q \wedge \forall m < n' + 1.\ \mathsf{follow}_m^f(p) \neq x)\ . \tag{8.68}$$

By splitting the universal quantification into the two parts $m = 0$ and $m \geq 1$, we obtain

$$\begin{aligned} \iff p = q \vee \exists n'.\ (\mathsf{follow}_{n'}^f(p\text{->}f) = q \wedge \\ p \neq x \wedge \forall 1 \leq m < n' + 1.\ \mathsf{follow}_m^f(p) \neq x)\ . \end{aligned} \tag{8.69}$$

The universal quantification is rewritten:

$$\begin{aligned} \iff p = q \vee \exists n'.\ (\mathsf{follow}_{n'}^f(p\text{->}f) = q \wedge \\ p \neq x \wedge \forall m < n'.\ \mathsf{follow}_m^f(p\text{->}f) \neq x)\ . \end{aligned} \tag{8.70}$$

As the first and the third conjunct are equivalent to the definition of $p\text{->}f \xrightarrow[x]{f} q$, the claim is shown. ∎

There are two simple consequences of rule ($A1$):

$$p \xrightarrow[x]{f} p \quad \text{and} \quad p \xrightarrow[p]{f} q \iff p = q\ . \tag{8.71}$$

In the following example we use these consequences to prove (8.61), the reachability predicate formula for our first verification condition.

Example 8.16. Recall (8.61):

$$list \overset{nxt}{\underset{0}{\rightarrow}} list \land \forall q \neq list.\ (list \overset{nxt}{\underset{list}{\rightarrow}} q \implies q\text{->}payload \neq a)\ . \tag{8.72}$$

The first conjunct is a trivial instance of the first consequence. To show the second conjunct, we introduce a **Skolem variable**[9] q' for the universal quantifier:

$$(q' \neq list \land list \overset{nxt}{\underset{list}{\rightarrow}} q') \implies q'\text{->}payload \neq a\ . \tag{8.73}$$

By the second consequence, the left-hand side of the implication is FALSE. ▟

Even when the axioms are used, however, reasoning about a reachability predicate remains tedious. The goal is therefore to devise an automatic decision procedure for a logic that includes a reachability predicate. We mention several decision procedures for logics with reachability predicates in the bibliographical notes.

8.6 Problems

8.6.1 Pointer Formulas

Problem 8.1 (semantics of pointer formulas). Determine if the following pointer logic formulas are valid using the semantic translation:

1. $x = y \implies \&x = \&y$.
2. $\&x \neq x$.
3. $\&x \neq \&y + i$.
4. $p \hookrightarrow x \implies *p = x$.
5. $p \hookrightarrow x \implies p\text{->}f = x$.
6. $(p_1 \hookrightarrow p_2, x_1 \land p_2 \hookrightarrow \text{NULL}, x_2) \implies p_1 \neq p_2$.

Problem 8.2 (modeling dynamically allocated data structures).

1. tt data structure is modeled by $\mathsf{my\text{-}ds}(q, l)$ in the following? Draw an example.

$$\begin{aligned}
\mathsf{c}(q,0) &\doteq (*q).p = \text{NULL} \\
\mathsf{c}(q,i) &\doteq (*\mathsf{list\text{-}elem}(q,i)).p = \mathsf{list\text{-}elem}(q,i-1) \quad \text{for } i \geq 1 \\
\mathsf{my\text{-}ds}(q,l) &\doteq \mathsf{list\text{-}elem}(q,l) = \text{NULL} \land \forall 0 \leq i < l.\ \mathsf{c}(q,i)
\end{aligned}$$

2. Write a recursive shorthand $\mathsf{DAG}(p)$ to denote that p points to the root of a directed acyclic graph.

[9] A Skolem variable is a ground variable introduced to eliminate a quantifier, i.e., $\forall x.P(x)$ is valid iff $P(x')$ is valid for a new variable x'. This is a special case of Skolemization, which is named after Thoralf Skolem.

3. Write a recursive shorthand $\mathsf{tree}(p)$ to denote that p points to the root of a tree.
4. Write a shorthand $\mathsf{hashtbl}(p)$ to denote that p points to an array of lists.

Problem 8.3 (extensions of the pointer logic). Consider a pointer logic that only permits a conjunction of predicates of the following form, where p is a pointer, and f_i, g_i are field identifiers:

$$\forall p.\ p\text{->}f_1\text{->}f_2\text{->}f_3\ldots = p\text{->}g_1\text{->}g_2\text{->}g_3\ldots$$

Show that this logic is Turing complete.

Problem 8.4 (axiomatization of the memory model). Define a set of memory model axioms for an architecture that uses 32-bit integers and little-endian byte ordering (this means that the least-significant byte has the lowest address in the word).

Problem 8.5 (partitioning the memory). Suppose that a pointer logic permits restrictions on L, the memory layout. Give a counterexample to Theorem 8.14.

8.6.2 Reachability Predicates

Problem 8.6 (semantics of reachability predicates). Determine the satisfiability of the following reachability predicate formulas:

1. $p \xrightarrow[p]{f} q \wedge p \neq q$.
2. $p \xrightarrow[x]{f} q \wedge p \xrightarrow[q]{f} x$.
3. $p \xrightarrow[q]{f} q \wedge q \xrightarrow[p]{f} p$.
4. $\neg(p \xrightarrow[q]{f} q) \wedge \neg(q \xrightarrow[p]{f} p)$.

Problem 8.7 (modeling). Try to write reachability predicate formulas for the following scenarios:

1. p points to a cyclic list where the next field is nxt.
2. p points to a NULL-terminated, doubly linked list.
3. p points to the root of a binary tree. The names of the fields for the left and right subtrees are l and r, respectively.
4. p points to the root of a binary tree as above, and the leaves are connected to a cyclic list.
5. p and q point to NULL-terminated singly linked lists that do not share cells.

Problem 8.8 (decision procedures). Build a decision procedure for a conjunction of atoms that have the form $p \underset{q}{\xrightarrow{f}} q$ (or its negation).

Problem 8.9 (program verification). Write a code fragment that removes an element from a singly linked list, and provide the verification conditions using reachability predicate formulas.

8.7 Bibliographic Notes

The view of pointers as indices into a global array is commonplace, and similarly so is the identification of structure components with arrays. Leino's thesis is an instance of recent work applying this approach [181], and resembles our Sect. 8.3. An alternative point of view was proposed by Burstall: each component introduces an array, where the array indices are the addresses of the structures [60].

Transitive closure is frequently used to model recursive data structures. Immerman et al. explored the impact of adding transitive closure to a given logic. They showed that already very weak logics became undecidable as soon as transitive closure was added [153].

The PALE (Pointer Assertion Logic Engine) toolkit, implemented by Anders Møller, uses a graph representation for various dynamically allocated data structures. The graphs are translated into monadic second-order logic and passed to MONA, a decision procedure for this logic [198]. Michael Rabin proved in 1969 that the monadic second-order theory of trees was decidable [236].

The reachability predicate discussed in Sect. 8.5 was introduced by Greg Nelson [204]. This 1983 paper stated that the question of whether the set of (eight) axioms provided was complete remained open. A technical report gives a decision procedure for a conjunction of reachability predicates, which implies the existence of a complete axiomatization [207]. The procedure has linear time complexity.

Numerous modern logics are based on this idea. For example, Lahiri and Qadeer proposed two logics based on the idea of reachability predicates, and offered effective decision procedures [175, 176]. The decision procedure for [176] was based on a recent SMT solver.

Alain Deutsch [102] introduced an alias analysis algorithm that uses *symbolic access paths*, i.e., expressions that symbolically describe which field to follow for a given number of times. Symbolic access paths are therefore a generalization of the technique we described in Sect. 8.5. Symbolic access paths are very expressive when combined with an expressive logic for the *basis* of the access path, but this combination often results in undecidability.

Benedikt et al. [24] defined a logic for linked data structures. This logic uses constraints on paths (called *routing expressions*) in order to define memory regions, and permits one to reason about sharing and reachability within such regions. These authors showed the logic to be decidable using a small-model property argument, but did not provide an efficient decision procedure.

A major technique for analyzing dynamically allocated data structures is *parametric shape analysis*, introduced by Sagiv, Reps, and Wilhelm [240, 251, 282]. An important concept in the shape analysis of Sagiv et al. is the use of Kleene's three-valued logic for distinguishing predicates that are true, false, or *unknown* in a particular abstract state. The resulting concretizations are more precise than an abstraction using traditional, two-valued logic.

Separation logic (see the aside on this subject on p. 184) was introduced by John Reynolds as an intuitionistic way of reasoning about dynamically allocated data structures [242]. Calcagno et al. [64] showed that deciding the validity of a formula in separation logic, even if robbed of its characteristic separating conjunction, was not recursively enumerable. On the other hand, they showed that, once quantifiers were prohibited, validity became decidable. Decidable fragments of separation logic have been studied, for example, by Berdine et al. [25, 26, 27]; these are typically restricted to predicates over lists. Parkinson and Bierman address the problem of modular reasoning about programs using separation logic [217].

Kuncak and Rinard introduced *regular graph constraints* as a representation of heaps. They showed that satisfiability of such heap summary graphs was decidable, whereas entailment was not [172].

Alias analysis techniques have also been integrated directly into verification algorithms. Manevich et al. described predicate abstraction techniques for singly linked lists [187]. Beyer et al. described how to combine a predicate abstraction tool that implements lazy abstraction with shape analysis [28]. Podelski and Wies propose *Boolean heaps* as an abstract model for heap-manipulating programs [230]. Here, the abstract domain is spanned by a vector of arbitrary first-order predicates characterizing the heap. Bingham and Rakamarić [36] also proposed to extend predicate abstraction with predicates designated to describe the heap. Distefano et al. [103] defined an abstract domain that is based on predicates drawn from separation logic. Berdine et al. use separation logic predicates in an add-on to Microsoft's SLAM device driver verifier, called TERMINATOR, in order to prove that loops iterating over dynamically allocated data structures terminate. A graph-based decision procedure for heaps that relies on a small-model property is given in [85]. The procedure is able to reason about the length of list segments.

Most frameworks for reasoning about dynamically allocated memory treat the heap as composed of disjoint memory fragments, and do not model accesses beyond these fragments using pointer arithmetic. Calcagno et al. introduced a variant of separation logic that permits reasoning about low-level programs including pointer arithmetic [63]. This logic permits the analysis of infrastructure usually assumed to exist at higher abstraction layers, e.g., the code that implements the `malloc` function.

8.8 Glossary

The following symbols were used in this chapter:

Symbol	**Refers to ...**	**First used on page ...**
A	Set of addresses	174
D	Set of data words	174
M	Map from addresses to data words	174
L	Memory layout	174
$\sigma(v)$	The size of v	174
V	Set of variables	174
$[\![\cdot]\!]$	Semantics of pointer expressions	179
$p \hookrightarrow z$	p points to a variable with value z	179
p->f	Shorthand for $(*p).f$	181
$\mathsf{list}(p, l)$	p points to a list of length l	182