Abstract. Theories of programming languages formalize pointers by formalizing the addresses, the heap and the stack of a computer storage. These are implementation concepts. The aim of this paper is a theory that formalizes pointers in terms of concepts from high-level programming languages. We begin with a graph theory, which formalizes the implementation concepts but avoids some common distinctions. From it, we calculate the theory of trace equivalences, which formalizes concepts of high-level programming languages. From that theory, we calculate definitions in terms of weakest (liberal) preconditions. We consider the assignment and the copy operation, which is introduced in the paper; the object creation (i.e. the new-statement) is a sequential composition of them. Those wlp/wp-definitions and the concept of trace equivalence are the result of the paper. They are intended as a foundation for program design; in particular, for an object-oriented one.

0 Introduction

By pointer theory, I mean a mathematical theory that formalizes pointers. Diverse pointer theories can be found in the literature (e.g. [1, 2, 11–13, 15–18]). In spite of their diversity, they have two characteristics in common.

First, these pointer theories formalize the addresses, the heap and the stack of a computer storage. Apart from differences in the mathematical concepts, the formalization is as follows. The stack holds the directly accessible program variables: it maps them into values. Among the values are integers, booleans, etc. and addresses. The heap holds the program variables that can be accessed only through a sequence of pointers: it maps pairs of an address and such a program variable into values. As an example, consider singly-linked lists. For any list element, let program variable data contain its integer value, and let program variable next point to the next list element. Let $x$ be a directly accessible program variable that points to a singly-linked list with at least two elements. The access to the integer value of the second list element is described as follows:

The stack maps $x$ to an address $m$. At address $m$ of the heap begins the first list element. The heap maps $m$ and next to an address $n$. At address $n$ of the heap begins the second list element. The heap maps $n$ and data to an integer.

In this way, pointers are formalized by addresses of the heap.
But addresses, the heap and the stack are not concepts of a high-level programming language. They belong to its implementation. Instead of (0), we write in the programming language (for example, in an object-oriented one)

\[ x.\text{next}.\text{data} \]  

(1)

Description (1) is much simpler than (0). But when we use a pointer theory, we must think as in (0).

In contrast, consider control structures, for example the while-loop. When we design a while-loop, we do not think about the labels and jumps by which it is implemented. Thanks to Dijkstra’s predicate-transformer semantics and to Hoare’s rule for the while-loop, we can think about it much more abstractly.

Second, pointer theories make several distinctions. They distinguish pointer \( \text{nil} \) from the other pointers. Since it points to no object, they do not formalize \( \text{nil} \) by an address. This distinguishes it from all other pointers.

Pointer \( \text{nil} \) must not be followed. When it is followed in an expression, pointer theories assign an undefined value to that expression. The undefined value is yet another value of pointers: it is not an address, nor is it equal to \( \text{nil} \).

Another distinction concerns values such as integers, booleans etc. Pointer theories distinguish them from pointers so that they must be dealt with separately. These three distinctions complicate our thinking about pointers.

Pointer theories are a foundation for program design. In particular, they are a foundation for object-oriented programming. (Throughout this paper, ‘address’ can be read as ‘object identifier’ and ‘heap’ as ‘object environment’.) Simplicity of pointer theories is therefore a major concern.

In this paper, the task is to devise a pointer theory that is free of the two characteristics discussed above: it must not formalize addresses, the heap and the stack, and it must avoid the three distinctions. It is intended as a foundation for program design. In contrast to the pointer theories in the literature, it still lacks predicates and theorems that are tailored to program design.

We must ensure that the pointer theory for which we are heading formalizes the usual idea of pointers and pointer operations. To ensure this, we begin with a pointer theory that formalizes addresses, the heap and the stack. That theory specializes Hoare’s and He’s graph theory \([9,10]\). The specialization eliminates the above three distinctions. In our graph theory, we define assignment, object creation and an operation called copy operation. We introduce the copy operation because it is simpler than the object creation, and the object creation can readily be expressed by it and the assignment. Therefore we can concentrate on the copy operation instead of the object creation.

Then we eliminate addresses, the heap and the stack from the graph theory. A theory results that we call the theory of trace equivalences. The assignment and the copy operation are redefined in the new theory. These new definitions are calculated from the graph-theoretic ones.

From the new definitions of the assignment and the copy operation, we calculate definitions in terms of weakest liberal preconditions and weakest preconditions. They conclude the derivation of our pointer theory.