Stanford Department of Computer Science
Report No. STAN-CS-80-779

September 1980

PROBLEMATIC FEATURES OF PROGRAMMING LANGUAGES: A SITUATIONAL-CALCULUS APPROACH

by

Zohar Manna and Richard Waldinger

Research sponsored by

National Science Foundation
Office of Naval Research
Department of the Air Force

COMPUTER SCIENCE DEPARTMENT
Stanford University

PROBLEMATIC FEATURES OF PROGRAMMING LANGUAGES: A SITUATIONAL-CALCULUS APPROACH PART I: ASSIGNMENT STATEMENTS

by

Zohar Manna
Stanford University
and Weizmann Institute

Richard Waldinger
SRI International

This research was supported in part by the National Science Foundation under Grants MCS-78-02591 and MCS-79-09495, in part by the Office of Naval Research under Contracts N00014-75-C-0816 and N00014-76-C-0687, and in part by the Air Force Office of Scientific Research under Contract F30602-78-0099.

The authors' addresses: Z. Manna, Department of Computer Science, Stanford University, Stanford, CA 94305; R. Waldinger, Artificial Intelligence Center, SRI International, Menlo Park, CA 94025.

ABSTRACT

Certain features of programming languages, such as data structure operations and procedure call mechanisms, have been found to resist formalization by classical techniques. An alternate approach is presented, based on a "situational calculus," which makes explicit reference to the states of a computation. For each state, a distinction is, drawn between an expression, its value, and the location of the value.

Within this conceptual framework, the features of a programming language can be described axiomatically. Programs in the language can then be synthesized, executed, verified, or transformed by performing deductions in this axiomatic system. Properties of entire classes of programs, and of programming languages, can also be expressed and proved in this way. The approach is amenable to machine implementation.

In a situational-calculus formalism it is possible to model precisely many "problematic" features of programming languages, including operations on such data structures as arrays, pointers, lists, and records, and such procedure call mechanisms as call-by-reference, call-by-value, and call-by-name. No particular obstacle is presented by aliasing between variables, by declarations, or by recursive procedures.

The paper is divided into three parts, focusing respectively on the assignment statement, on data structure operations, and on procedure call mechanisms. In this first part, we introduce the conceptual framework to be applied throughout and present the axiomatic definition of the assignment statement. If suitable restrictions on the programming language are imposed, the well-known Hoare assignment axiom can then be proved as a theorem. However, our definition can also describe the assignment statement of unrestricted programming languages, for which the Hoare axiom does not hold.

1. INTRODUCTION

The most widely accepted approach to program verification has been the one described in Floyd's [1967] paper and formalized by Hoare [1969]. Hoare's formalization requires that each construct of the programming language be described by an axiom or rule, specifying how the construct alters the truth of an arbitrary assertion. Certain features of programming languages have been found to be easier to describe in this way than others:

- Programs with only simple assignment statements and **while** statements can be described adequately.

- Programs with arrays are less tractable, but can be treated if the array operations are rewritten in terms of of McCarthy's [1962] **assign** and **contents** functions.

- Operations on other data structures, such as pointers, lists, and records, can be handled only if special restrictions are imposed on the language.

- Different varieties of procedure calls have also required programming-language restrictions.

- Even the simple assignment statement fails to satisfy the usual Hoare assignment axiom if included in a programming language with other problematic features.

- Certain combinations of features have been shown (Clarke [1977]) to be impossible to describe at all with a Hoare-style axiomatization.

It has been argued (e.g., in London, R. L. et *al.* [1978]) that features of programming languages whose semantics are difficult to describe with the Floyd-Hoare technique are also difficult for people to understand and use consistently. For this reason, a number of programming languages have been designed with the intention of eliminating or restricting such "problematic" features. Others have objected (e.g., Knuth [1974], Hoare [1975], deMillo et *al.* [1979]) that the disciplined use of such "unverifiable" programming features can facilitate the clear and direct representation of a desired algorithm, while their removal may force the programmer into increasingly obscure circumlocutions.

In this paper, we present a conceptual framework capable of describing all these problematic programming features. This framework is suitable to serve as a basis for the implementation of verification systems, as well as synthesis and transformation systems. We do not argue that the problematic features should necessarily be included in programming languages without restriction, but we intend that if a language designer wishes to use some combination of features, no obstacle should be imposed by verification concerns.

The approach we employ is a "situational calculus," in which we refer explicitly to the states of the computation. In a given state s, the evaluation of an expression e of the programming language produces a new state s; e. The meaning of the expression can then be defined by axioms that relate the characteristics of the new state to those of the original state.

This formalism is quite distinct from that of Hoare, in which no explicit reference is made to states. In this respect, our approach is closer to those adopted by McCarthy [1964] and Burstall [1969] for specifying the semantics of ALGOL-60 subsets, and by Green [1969] for describing robot operations.

To describe the characteristics of the states of a computation, we introduce "situational operators," i.e., functions and predicates whose values depend on the state. In defining these operators, we distinguish between the expressions of the programming language, the storage locations of the machine, and the abstract data objects of the program's domain. The precision of this descriptive apparatus enables us to model the effects of programming-language constructs in full detail. We can describe and compare various implementations of the same programming language or, if we prefer, we can ignore the details of implementation.

Once we have succeeded in describing the constructs of a programming language, we can use that description in proving that programs in the language satisfy a given specification. The situational operators can be used not only to describe the constructs of the language but also to represent the specifications of a program. Indeed, they are more expressive for this purpose than the conventional input/output assertions, because they enable us to refer in a single sentence to different states of the computation. For example, it is possible to say directly how the *final* value of an identifier relates to its initial or intermediate values. To show that a program satisfies such a specification, we then prove a corresponding theorem in situational calculus.

The situational-calculus approach can be applied not only to prove that a single program satisfies given properties, but also to prove that an entire class of programs, or a programming language, satisfies given properties. For example, we can state and prove that the "aliasing" phenomenon, in which two identifiers are different names for the same location, cannot be created in languages that satisfy certain constraints.

Although the approach has been devised to treat languages for which the Hoare formalism breaks down, it can also be used to show that the Hoare formalism does actually apply to suitably restricted programming languages. For example, we can show that the Hoare assignment axiom (which fails to apply to most languages used in practice) is true and can be proved as a situational-calculus theorem for languages in which the problematic features have been omitted.

Up to now, we have been discussing the use of a situational-calculus approach for proving properties of given programs and classes of programs. Historically, however, we were led to this approach in developing a method for program synthesis, i.e., the systematic construction of a program to meet given specifications. We have described elsewhere (Manna and Waldinger [1980]) a deductive technique for the synthesis of applicative programs, i.e., programs that yield an output but produce no side effects. We can now construct programs that may produce side effects by applying the same deductive technique within the situational calculus. More precisely, to construct a program to achieve a desired condition, we prove the existence of a state in which the condition is true. The proof is constrained to be "constructive," so that a program to achieve the desired condition can then be extracted from the proof.

The same deductive technique can be applied to the task of transforming a given program, generally to improve its efficiency. Often the performance of a program can be augmented, at the expense of clarity, by applying transformations that introduce side effects. This transformation process can be conducted within a situational-calculus deductive system to ensure that the original purpose of the given program is preserved. For example, a purely applicative program for reversing a list can be transformed into one that achieves the same purpose with side effects, by altering the structure of the list.

The situational calculus is a more appropriate logic of programs than dynamic logic (Pratt [1976]), because it is able to describe the results of evaluating nested expressions as well as the effects of executing sequences of statements: dynamic logic is not intended to apply to expression-oriented languages. The approach of this paper is more similar in intent and scope to that of denotational semantics, but ours relies on a simpler mathematical framework. We do not use functions of higher type, lambda expressions, or fixed points. Situational calculus can be embedded comfortably in a first-order logic to which the well-developed battery of mechanical theorem-proving techniques, such as the unification algorithm, can be applied. In particular, no special difficulty is presented by the existential quantifier, which is outside the scope of denotational semantics-based systems (e.g., Gordon et *al.* [1979]), but which is valuable for program verification and crucial for program synthesis.

The paper is divided into three parts. The present part (Part I) introduces the general conceptual framework and discusses its application to the assignment statement. Future parts describe the application of the same conceptual framework to the description of data structure operations and procedure call mechanisms.

We begin (in *Motivation*) by describing some of the features of programming languages that have caused problems in the past, and indicate sources of the difficulty. We then introduce (in Conceptual Framework) the situational operators that define the characteristics of states. The conceptual framework is then used (in The Assignment Statement) to express the axioms that describe the behavior of the simple assignment statement. We show (in *Howe's Assignment Axiom*) that the statement usually taken as the assignment axiom can be expressed and, for suitably restricted programming languages, proved as a theorem in our formalism. Finally (in *Assignment Expressions*), we apply the conceptual framework to a wider class of programs, which violate the Hoare axiom.

In the forthcoming Part II, we apply the technique to describe data structures and the operations that manipulate them. We introduce a general notion of data structure, which includes arrays, pointers, trees, and records, and we define an operation for altering such a data structure. A single set of axioms describes the behavior of this operation. We allow different data structures to share substructures, and we do not exclude circular structures.

In the forthcoming Part III, we use our situational-calculus framework to describe declarations and procedure calls. We consider several different calling mechanisms, including call-by-reference, call-by-value, and call-by-name, and we admit both static and dynamic binding of global identifiers. We allow procedures to be passed as arguments to other procedures. No special problems are presented by recursive calls.

We will not be concerned in this paper with the problems presented by computer arithmetic, array bounds, types, coercion, or error recovery, although these can be described in our formalism with no special difficulty. We will also ignore the phenomena of parallelism and nondeterminism; these require substantive extension of the framework.

2. MOTIVATION

In this section, we discuss some features of programming languages that have proved difficult to formalize in the classical framework, and we outline some of the solutions that have been proposed. In the following section we introduce a general framework that uniformly resolves all these difficulties.

We begin with one of the less problematic features — the simple assignment statement

A. Assignment to Identifiers

By the simple assignment statement we mean one of the form

$$\mathbf{x} \leftarrow \mathbf{t},$$

where x is an identifier and **t** is an expression in the programming language. The **Hoare** axiom for such an assignment may be expressed as

$$\{P \triangleleft \langle \mathbf{x} \leftarrow \mathrm{t} \rangle\}\ \mathbf{x} \leftarrow \mathbf{t}\ \{P\},$$

indicating that if the assertion $P \triangleleft \langle \mathbf{x} \leftarrow \mathrm{t} \rangle$ holds before executing the assignment $\mathbf{x} \leftarrow \mathbf{t}$, and if the execution terminates, then the assertion P holds afterward. Here, $P \triangleleft \langle \mathbf{x} \leftarrow \mathbf{t} \rangle$ is the result of replacing all free **occurences** of $\mathbf{x}$ in P by $\mathbf{t}$. The rationale for this rule is that the value of x after executing the assignment $\mathrm{x} \leftarrow \mathbf{t}$ will be the same as the value of **t** before; therefore, anything that can be said about **t** before the execution can be said about x afterwards.

However, the above reasoning is faulty, and only applies if certain restrict ions are applied to the expression **t**, the assertion *P*, and the situation in which the assignment takes place. Let us examine some of these restrictions:

• The expression t must be *static*, in the sense that its evaluation must not itself produce side effects. For example, in the assignment

$$\mathbf{x} \leftarrow (\mathbf{x} + (\mathbf{y} \leftarrow \mathbf{y+1})),$$

the evaluation of **t**, that is, $\mathbf{x+(y \leftarrow y+1)}$, has the side effect of altering the value of the identifier y. Such assignments are legal in the ALGOL dialects and in

LISP. If we take the assertion P to be $y = 0$, then according to the Hoare axiom, we have

$$\{y = 0\}\ x \leftarrow (x + (y \leftarrow y+1))\ \{y = 0\}.$$

(Note that the assertion $(y = 0) \triangleleft \langle x \leftarrow t \rangle$ reduces to $y = 0$ because x does not occur in $y = 0$.) However, this sentence is false because, if y is 0 before executing this assignment, then afterwards y will be 1, not 0.

Similarly, the assignment

$$x \leftarrow f(x),$$

where f is a procedure that has the side effect of increasing the value of the global identifier y by 1, violates the instance of the Hoare axiom

$$\{y = 0\}\ x \leftarrow f(x)\ \{y = 0\}.$$

• In the situation in which the assignment to the identifier x takes place, there must be no way to refer to x indirectly, in terms of other identifiers. For example, suppose x and y are "aliases," i.e., they are different names for the same location. Then changing the value of x will also change the value of y. If P is the condition $\{y = 0\}$, then the instance of the Hoare axiom

$$\{y = 0\}\ x \leftarrow 1\ \{y = 0\}$$

is false because, after executing the assignment, the value of y will be 1, not 0.

In practice, the aliasing phenomenon can arise in languages that admit procedure calls. For example, suppose we have a procedure

$$f(x, y) \Leftarrow x \leftarrow 1$$

whose parameters are passed by a call-by-reference mechanism. In other words, in executing a procedure call $f(u, v)$, where u and v are identifiers, the identifiers x and u become aliases, and the identifiers y and v become aliases. In executing the procedure call $f(u, u)$, all three identifiers x, y, and u become aliases, so altering the value of x will alter the value of y as well. Thus, the assignment statement $x \leftarrow 1$ that occurs in the body of the procedure $f(x, y)$ will violate the instance of the Hoare axiom

$$\{y = 0\}\ x \leftarrow 1\ \{y = 0\}.$$

Aliasing can occur in other ways besides the action of the procedure call mechanism. In FORTRAN an alias can be created directly by the action of the **"common"** or "equivalence" statements.

Other situations, aside from simple aliasing, in which the **Hoare** axiom is violated occur when the identifier x is bound to a substructure of some data structure. Then altering the value of **x** will indirectly alter the value of the structure.

• The assertion *P* may not refer to the value of an identifier except by mentioning the identifier explicitly. For example, suppose *P* is the assertion

"there exists an identifier whose value is 2."

Because *P* contains no occurrences of x at all, the sentence

$$\{P\}\ \mathrm{x} \leftarrow 0\ \{P\}$$

is an instance of the **Hoare** axiom. However, if x is the only identifier whose value is 2 before executing the assignment, then *P* will become false afterwards. Here, the axiom broke down because *P* referred to the value of x without mentioning x itself.

Many of the problems connected with the assignment statement, and **a denotational** approach to their solution, are described in Ligler [1975].

B. Array Assignments

The direct translation of the **Hoare** assignmentaxiom to array assignments is

$$\{P \triangleleft (\mathrm{a[x]} \leftarrow \mathrm{t})\}\ \mathrm{a[x]} \leftarrow \mathrm{t}\ \{P\}.$$

This sentence is false, even for straightforward expressions **t** and assertions ***P***, and for the simplest situations. For example, the sentence

$$\{\mathrm{a[y]} = 0\}\ \mathrm{a[x]} \leftarrow 1\ \{\mathrm{a[y]} = 0\}$$

is an instance of the above sentence, because **a[x]** does not occur at all in the assertion **a[y]** = 0; but of course the sentence is false if x and y have the same values. The problem is that, while it is exceptional for two identifiers x and y to be aliases, it is commonplace for two array entries a[x] and a[y] to be alternative names for the same location.

The difficulty has been approached (McCarthy [1962]) by regarding the entire array as a single entity, so that assigning to any of the array's entries produces a new array. More precisely, we regard the entire array a as an ordinary identifier, we treat an array assignment a[x] ← t as an abbreviation for a simple assignment

$$a \leftarrow assign(a, x, t),$$

and we treat an array access a[y] as an abbreviation for

$$contents(a, y).$$

The assign and contents functions are then assumed to satisfy the properties

$$contents(assign(a, x, t), y) = t \qquad \text{if } x = y$$

and

$$contents(assign(a, x, t), y) = contents(a, y) \qquad \text{if } x \neq y.$$

Programs involving arrays can then be treated by the Hoare axiom for simple identifier assignments.

Thus, the previous false sentence

$$\{a[y] = 0\}\ a[x] \leftarrow t\ \{a[y] = 0\},$$

expressed in terms of the contents and assign functions, is

$$\{contents(a, y) = 0\}\ a \leftarrow assign(a, x, t)\ \{contents(a, y) = 0\}.$$

This sentence is not an instance of the Hoare assignment axiom, because the assertion contents(a, y) = 0 does contain an occurrence of the identifier a. The appropriate instance of the Hoare axiom in this case is the true sentence

$$\begin{aligned}&\{(contents(a, y) = 0) \lhd (a \leftarrow assign(a, x, t))\}\\&a \leftarrow assign(a, x, t)\\&\{contents(a, y) = 0\},\end{aligned}$$

i.e.,

```
{contents(assign(a, x, t), y) = 0}
a ← assign(a, x, t)
{contents(a, y) = 0}.
```

Although this solution still suffers from the limitations associated with the simple assignment axiom, it resolves the special difficulties arising from the introduction of arrays.

C. Pointer Assignment

To describe the pointer mechanism, let us introduce some terminology. If an identifier is declared in a program, there is some location bound to that identifier; we can regard the location as a cell in the machine memory. If two identifiers are aliases, they are bound to the same location. A location may contain data or it may store (the address of) another location; we thus distinguish between data *locations* and storage locations. A pointer is a storage location that stores (the address of) another storage location. There are many notations for pointers in different programming languages; ours is typical but is not actually identical to any of these.

A pointer is created, for example, by the execution of the simple assignment

$$x \leftarrow \uparrow y,$$

where x and y are both identifiers. Here the notation $\uparrow y$ means the location bound to the identifier y. The result of this assignment is that the location bound to y is stored in the location bound to x. The configuration produced may be represented by the following diagram:

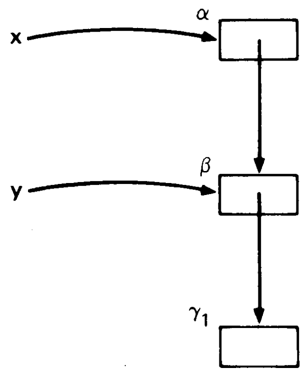


Figure 2.1

Here, α and β are locations bound to x and **y,** respectively; γ_1 is the location stored in β.

If we subsequently execute a simple assignment statement

$y \leftarrow t,$

where **t** is an expression, we alter the contents of the location β that y is bound to. The location β will then store the location γ_2 yielded by the evaluation of **t.** The new configuration can be represented by the following diagram:

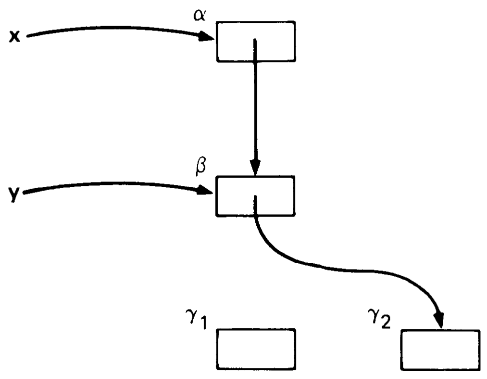


Figure 2.2

We have remarked that such a configuration can easily lead to violations of the **Hoare** assignment axiom: a simple assignment to y can alter the truth of an assertion about x.

Suppose instead we execute the special pointer *assignment*

$$\downarrow x \leftarrow t.$$

The notation $\downarrow x$ means that the location altered by the assignment is not the location α bound to x, but rather the location β stored in α. In other words, the effect of the above pointer assignment is identical to that of the simple assignment $y \leftarrow t$, and results in the same configuration depicted above.

A naive extension of the **Hoare** axiom to pointer assignments is

$$\{P \lhd (\downarrow x \leftarrow t)\} \downarrow x \leftarrow t \{P\}.$$

The sentence

$$\{y = 0\} \downarrow x \leftarrow 1 \{y = 0\}$$

is an instance of this axiom, because $\downarrow x$ does not occur in the assertion $y = 0$. However, as we have seen, if x "points to" y the assignment $\downarrow x \leftarrow 1$ can set the value of y to 1. In short, the simple adaptation of the **Hoare** assignment axiom fails to describe the action of the pointer assignment; because the assignment can alter the value of an identifier not mentioned explicitly.

The **assign/contents** technique for arrays has been extended (e.g., see Cart wright and Oppen [1978]) to pointers by regarding all the identifiers in the program as entries in a single array **v**, which is indexed not by integers, but by identifiers. These array operations can then be treated as simple assignments in terms of the **assign** and **contents** functions, and are correctly described by the **Hoare** simple assignment axiom and the two McCarthy axioms for **assign** and **contents**.

The **Hoare** formalism itself was extended by Schwartz and Berry [1979] to handle pointers and other constructs of ALGOL 68. This approach employs two distinct mechanisms to refer to the states of a computation: the **Hoare** $P\{F\}Q$ operator and explicit situational operators.

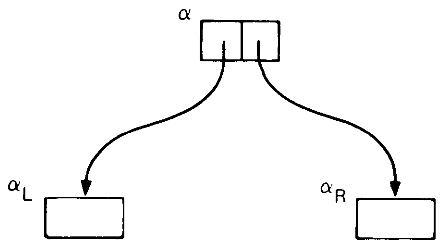
D. Tree and Record Structure Manipulation

The problems involved with arrays and pointers are compounded in languages with facilities for manipulating more complex structures, such as trees and records. Up to now we have considered only locations that contain data or that store (the address of) a single other location. To treat general data structures, we introduce locations that can store (the address of) arbitrarily many other locations.

For example, in LISP we admit binary-tree *locations,* which can store precisely two locations. If α is a binary-tree location, we will call the two locations stored in α the *left* and right *descendants* of a. Of course, these locations may themselves be binary-tree locations. We will refer to the descendants of α as a set that includes not only α**'s** left and right descendants, but also their left and right descendents, and so on; we will say that α is one of their *ancestors.*

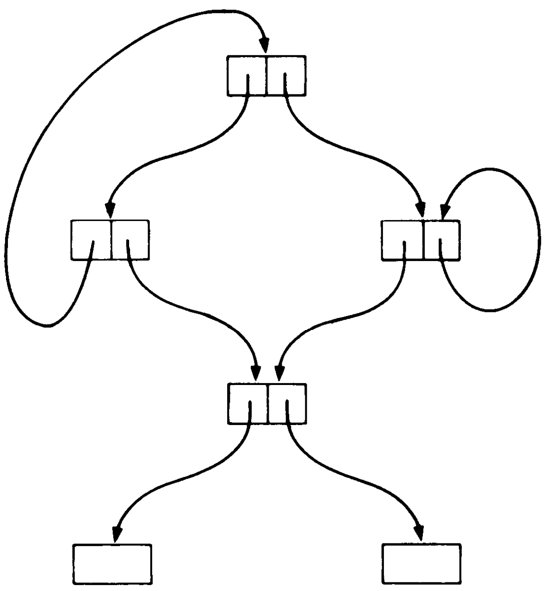
We will depict a binary-tree location α and its left and right descendents α_L and α_R by the following diagram:

Figure 2.3



We do not exclude the possibility that a binary-tree location is identical to any of its own descendants; such a configuration is called a circular tree. For example, the tree below is circular:

Figure 2.4



LISP provides two functions, which we will call **left** and **right**, for accessing the corresponding descendants of a binary-tree location, Suppose *t* is an expression whose evaluation yields a binary-tree location α; in other words, α represents the value of *t*. Then the evaluation of **left(t)** and **right(t)** yields the left and right descendants of α, respectively.

LISP also provides two operations for altering binary trees: the rpfucu operation, which we denote by

left(e) ← t,

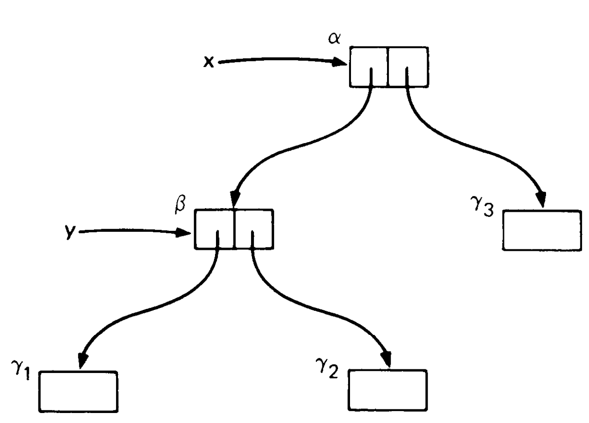
and the *rplacd* operation, which we denote by

right(e) ← t,

where e and **t** are any expressions. If the evaluation of e yields a binary-tree location α, and if the evaluation of **t** then yields a location β, then the *rplaca* operation **left(e) ←t** will cause β to become the new left descendant of at. The *rplacd* operation behaves analogously.

The problem in describing the *rplaca* and *rplacd* operations is precisely the same as for the pointer assignment: a *rplaca* operation on one binary tree can alter the value of another without mentioning it explicitly. For example, suppose that x and y are identifiers associated with binary-tree locations α and β, respectively, in the following configuration:

Figure 2.5



If *we* execute the *rpfucu* operation **left(y)** ← t, we obtain the configuration

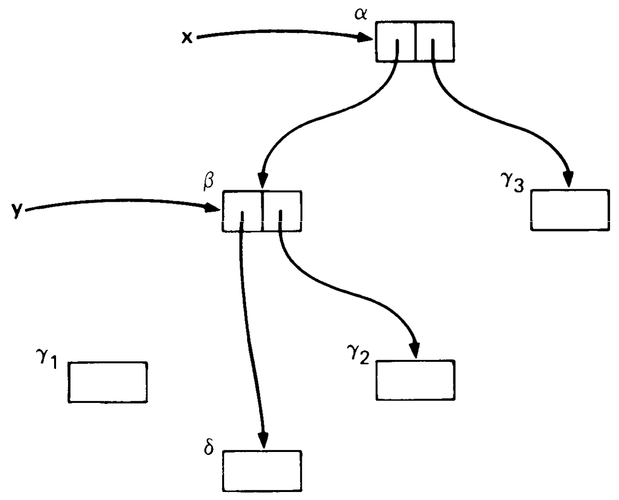


Figure 2.6

where δ is the location yielded by the evaluation of **t**. Note that the subsequent evaluation of the expression **left(left(x))** yields the location δ, not the location γ_1. In other words, the value of x may have been changed by the *rpfucu* operation, even though the operation was applied to y, not to x. Similarly, if α had any other ancestors before the execution of the *rplaca*, then their values could also have been affected by the operation. It seems that, to model the effects of such an assignment completely, we must know all of the ancestors of the altered location.

Many languages admit a more general form of tree structure called a record, in which a record *locution* can store several other locations. Binary trees can then be regarded as a special type of record. The same problems that arise with trees clearly come up with records as well.

The **assign/contents** formalization of arrays has been extended to apply to tree and record structures by Wegbreit and Poupon [1972], Cartwright and Oppen [1978], Levy [1978], and Kowaltowski [1979]. Burstall [1972] represents the operations that alter tree and record structures by introducing new functions to access the structures. For example, an *rpfucu* operation is said to create a new access function **left'**, which behaves like the **left** function after the execution of the assignment.

E. Expressiveness of Specifications

Many of the difficulties that prevent us from describing the behavior of individual programming constructs with the Floyd/Hoare approach also impede our efforts to express the specifications that describe the desired behavior of entire programs. The only mechanism for forming specifications in that approach is a pair of input and output assertions. We have already encountered one weakness in the expressive power of such assertions: there is no way to refer to an identifier without mentioning it explicitly. For example, we were unable to deal with an assertion such as

*there exists an identifier whose value is 2"

in which we refer to an identifier indirectly.

Other such limitations also restrict our ability to specify programs. For example, suppose we wish to state that a given procedure will behave properly if initially two of its input parameters x and y are not aliases of each other, i.e., they are not bound to the same location.

A natural approach might be to introduce the condition

$$not(alias(\mathbf{x}, y))$$

as part of the program's input assertions and to describe the relation $alias(\mathbf{x}, \mathbf{y})$ with axioms or rules of inference. However, this relation cannot be expressed in an assertion, because x and y are meant to refer to locations, not values. Thus, the relation will violate even the simple assignment axioms; e.g., the instance of the **Hoare** axiom

$$\{alias(\mathbf{x}, y)\}\ z \leftarrow y\ \{alias(\mathbf{x}, z)\}$$

is false: if x and y are aliases, then assigning the value of y to z will not cause x and z to become aliases.

This shortcoming foils a plausible approach to retaining the **Hoare** formalism: forbidding aliasing to occur in situations where it can lead to trouble. We certainly can forbid such occurrences, but we cannot express the condition we want to forbid as a **Hoare** assertion.

Another awkward aspect of in the Floyd/Hoare assertion mechanism as a specification device is its inability to refer to more than one state in a single assertion. Thus, it is impossible in an output assertion to refer directly to the initial or intermediate value of an identifier. For example, suppose we want 'to say that a program reverses the values of two identifiers x and y. The traditional approach is to introduce a "ghost" input assertion

$$\{\mathbf{x} = \mathbf{x}_0 \text{ and } \mathbf{y} = \mathbf{y}_0\}$$

at the beginning of the program, so that at the end we can assert that

$$\{x = y_0 \text{ and } y = x_0\}.$$

The purpose of the input assertion is merely to give names to the initial values of x and y. We must be careful, of course, to ensure that x_0 and y_0 are new identifiers that do not occur in the program.

The flaw in this solution is apparent if we attempt to use the above program not in isolation but as a segment of a larger program or as the body of a procedure. In this case, we would normally have to prove that the initial assertion

$$\{x = x_0 \text{ and } y = y_0\}$$

is true when control enters the segment; but this is impossible, because x_0 and y_0 are new symbols that cannot occur earlier in the program. Special mechanisms are required for dealing with ghost assertions.

F. Procedures

We have already seen that procedure calls can cause aliasing to occur, which obstructs attempts to axiomatize the assignment statement; we have also seen how global side effects of procedure calls foil the assignment statement axiomatization. Many other problems arise in describing the procedure call mechanism itself. Let us consider only one of these difficulties: expressing how global identifiers of procedures are treated in languages with static binding.

A global *identifier* of a procedure is one that occurs in the procedure's body but is not one of its parameters. For example, consider the procedure f(z) declared by

$$f(x) \Leftarrow x \leftarrow x + y.$$

Here, y is a global identifier of f, but x is not.

In a language with static binding, such as PASCAL or the ALGOL dialects, the binding of y that would be used in evaluating the procedure is the binding that y had when the procedure was declared. On the other hand, in a language with dynamic binding, such as LISP, the binding used would be the one that y had when the procedure was called.

Static binding is difficult to treat by a Hoare rule for a procedure call because it requires that we refer to the binding the global identifier y had in a much earlier state, when the procedure f was first declared. In the meantime, of course, y may have been redeclared.

* * * * *

It is the intention of this paper to present a single conceptual framework capable of describing all of these problematic programming-language features. This framework is compatible with contemporary theorem-proving techniques and can be incorporated into systems for the synthesis, verification, and transformation of computer programs.

3. CONCEPTUAL FRAMEWORK

In this section we introduce the conceptual framework on which our descrip tion of programming-language constructs is based.

A. The Objects

We assume that the objects we consider include the following:

- A set S of states, including a special *undefined state* $\perp_s$.

- A set *PL* of *expressions* in a programming language. We will use boldface **(boldface)** symbols to denote expressions in *PL.*

- A set *L* of (machine) **locations,** including a special *undefined locution* $\perp_l$. The defined locations are divided into three disjoint sets:

  - the data **locutions,**
  - the *storage* **locutions,** and
  - the *structure locutions*

- A set *D* of *data objects,* including the logical objects true and *false,* **and** the *undefined* object $\perp_d$. The defined data objects include

  - the *atomic data objects* (or *atoms*).

The intuition that motivates these definitions is as follows:

A data object is an abstract mathematical entity, such as a number, a function, or a string. The atoms will be those data objects that are not considered to be composed of other data objects. Typically, integers and the truth values *true* and *false* will be atoms; lists, strings, and functions will be nonatomic data objects.

The data and structure locations are the machine representations of the **data** objects; the data locations represent the atomic data objects; the structure locations represent the nonatomic data objects. (We will not be discussing nonatomic data objects and structure locations in Part I of this paper.) A storage location is one that can store (or point *to)* another location.

We will not restrict **ourself** to any particular programming language, although from time to time we will introduce typical language features for discussion. Our intention is that the formalism should be expressive enough to describe a wide **class** of programming-language constructs.

Let us assume that the programming language *PL* contains a **class** of identifiers. We mean the identifiers to include those symbols, such as *2,* that are used as names for atomic' data objects. Furthermore, we assume that a sequencing operation **";"** is included in *PL*; thus if $e_1, e_2, \ldots, e_n$ are expressions, then $e_1; e_2; \ldots; e_n$ is an expression that denotes the program that evaluates first e_1, then $e_2, \ldots,$ and finally e,.

A state is **a** particular situation in the course of a computation. If s is *a* state and e is an expression, then $s; e$ denotes the result of evaluating the expression e in state s. If $e_1, e_2, \ldots,$ and e_n are several expressions, then $s; e_1; e_2; \ldots; e_n$ denotes the state that results if we evaluate $e_1, e_2, \ldots, e_n$ in sequence, beginning in state s. We will assume the ***sequence*** *property*

<3.1> $s;\ (ei;\ e_2) = (s;\ e_1);\ e_2,$

so that we can write $s; e_1; e_2$ without ambiguity. If the evaluation of e in state *8* leads to an error or does not terminate, then $s;$ e will turn out to be the undefined state $\perp_s$.

Furthermore, if s itself is undefined, then $s; e$ is also undefined, i.e.,

<3.2> $\perp_s; e \equiv \perp_s.$

B. Basic Situational Operators

Situational operators are functions or predicates, one of whose arguments is a state. They are used to denote entities that may change during a computation. We will use the following situational functions for any state s:

- for any identifier x,

$$loc(s, x)$$

is a location

- for any location ℓ,

$$store(s, \ell)$$

is also a location

- for any location ℓ,

$$data(s, \ell)$$

is a data object.

The intuition behind these definitions is that if x is an identifier bound to a machine location ℓ (e.g., via a symbol table), then $loc(s, \mathbf{x}) = \ell$. If ℓ_1 is a location that addresses (or stores) a location ℓ_2, then $store(s, \ell_1) = \ell_2$. If ℓ is a location that represents a data object d, then $data(s, \ell) = d$.

The situational functions may take undefined arguments and yield undefined values. We assume that the situational functions are ***strict***, in the sense that if any of their arguments are undefined, then their value is undefined too. For example, if s is the undefined state $\perp_s$, then $loc(s, \mathbf{x}) = \perp_\ell$. Furthermore, if the identifier x is not declared in a defined state s, then $loc(s, \mathbf{x}) = \perp_\ell$ as well. We assume that if ℓ is a data or structure location, then $store(s, \ell) = \perp_\ell$ in any state s; in addition, $store(s, \ell)$ may be undefined even if ℓ is a storage location in a defined state s.

We define three situational predicates. For any state s and location ℓ:

- $isdata(s, \ell)$ is true if ℓ is a data location in state s
- $isstore(s, \ell)$ is true if ℓ is a storage location in state s
- $isstructure(s, \ell)$ is true if ℓ is a structure location in state s

Note that the situational predicates are never undefined; if any of their arguments are undefined, they are false.

The relationship among the sets of data, storage, and structure locations can now be expressed in terms of these predicates:

<3.3> $isdata(s, \ell)$ ***or*** $isstore(s, \ell)$ ***or*** $isstructure(s, \ell)$

<3.4> ***not(isdata(s, ℓ)*** *and* ***isstore(s, ℓ))***

<3.5> ***not(isdata(s, ℓ)*** *and* ***isstructure(s, ℓ))***

<3.6> ***not(isstore(s, ℓ)*** *and* ***isstructure(s,*** *C))*

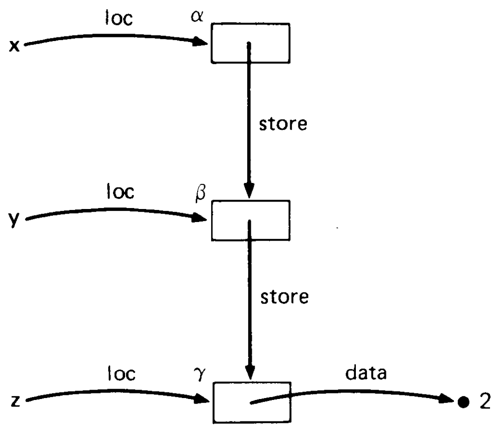
for all defined states s and defined locations ***ℓ***.

We assume that the *store* operator "transmits" *data; i.e.,*

<3.7> *data(s,* ***ℓ)*** = *data(s,* ***store(s, ℓ))*** if ***isstore(s, ℓ)***.

In other words, if one location stores another, we consider them to represent the same data object. For example, consider the following configuration:

Figure 3.1



Here, because

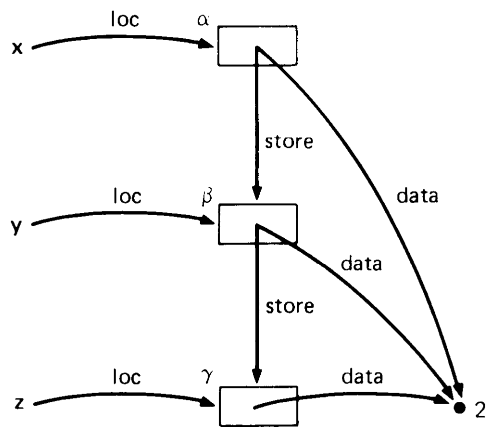
store(s, *a)* = ***β*** and ***store(s,β)*** = *7,*

we have

data(s, a) = *data(s,* ***β)*** = *data(s, 7)* = *2.*

Thus, we can conclude that the full diagram for this configuration would be

Figure 3.2



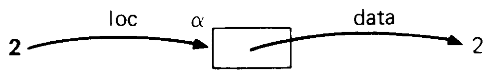
The definition of *data(s, ℓ)* for structure locations ℓ will be given in the forthcoming Part II. Now, let us give some examples to illustrate the use of these concepts.

Example *(constant):*

Assume that the programming language *PL* contains the identifier 2; then normally, this identifier would be bound to a data location that represents the data object 2.

This configuration is illustrated by the following diagram

Figure 3.3



If terms of our situational operators, we have

$$loc(s, 2) = a$$

$$data(s,\alpha) = 2.$$

where

$$isdata(s, a) \text{ is true.}$$

An assignment statement will be defined (in a later section) to alter the *store* link of a storage location. An identifier is bound to a data location rather than a storage location if we do not intend to alter its value by an assignment statement. Thus, numbers such as **1, 2.2,** -4, etc. and other special identifiers such as pi or nil might be bound in this way. In common parlance, such identiflers or their associated locations are called "constants"; however, we will avoid this terminology because it conflicts with the way the word **"constant"** is used in logic.

Ezample (variable):

Assume that the identifier **x** is bound to a storage location a, that α stores (addresses) a data location β, and that β represents the data object 2. (We will avoid using the word Variable," commonly applied to **x** or α, because of a conflict with logical usage.) The configuration we have described is illustrated by the following diagram:

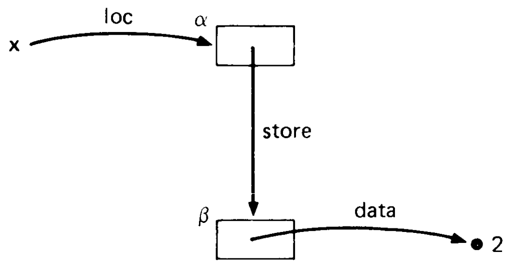


Figure 3.4

In terms of our situational operators, we have

$$loc(s, \mathbf{x}) = a$$
$$store(s, \alpha) = \beta$$
$$data(s, \beta) = 2$$

where

isstore(*s*, a) is true
isdata(*s*, *β*) is true.

Informally, we will refer to a situational function in a given state as *a* "link"; thus, we will say there is a store link between ***α*** and ***β*** in state ***s***.

c. Derived Situational Operators

There are some other situational operators that are defined in terms of the basic situational operators.

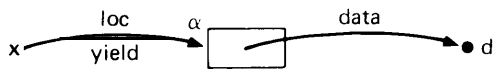
• The situational function *yield*(*s*, x) is the location yielded by the evaluation of an identifier **x** in state s. We will define it by

<3.8> $yield(s, x) = loc(s, x)$ if $isdata(s, loc(s, x))$
or $isstructure(s, loc(s, x))$

<3.9> $yield(s, x) = store(s, loc(s, x))$ if $isstore(s, loc(s, x))$

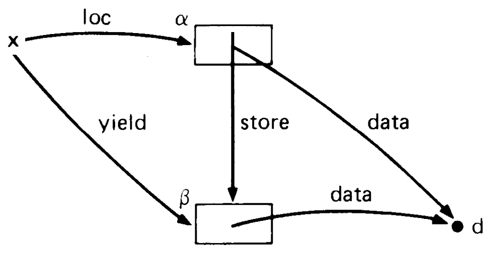
for all states s and identifiers x. In other words, if **x** is bound to a **data** or structure location, then the evaluation of x yields that location. On the other hand, if x is bound to a storage location, then the evaluation of **x** yields the location stored in that storage location. Thus, the diagrams that illustrate these two situations are

Figure 3.5



and

Figure 3.6

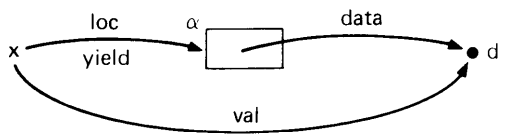


• The situational function *val(s, x)* *is* the value of **x** in state *s*. We define it to be the data object represented by *yield(s, x)*; *i.e.,*

<3.10> $val(s, x) = data(s, yield(s, x))$

for all states s and identifiers x. Depending on whether *loc(s,* x) is *a* data or structure location or a storage location, the *val* operator is illustrated by one of the following two diagrams:

Figure 3.7



and

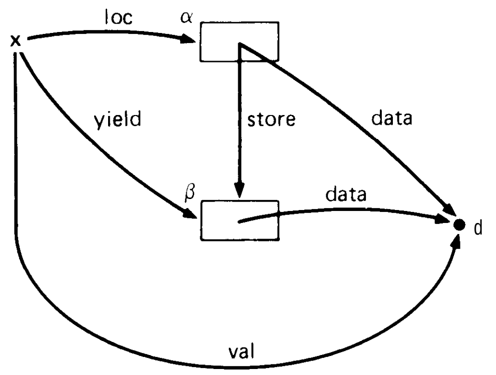


Figure 3.8

In the first diagram, *α* is a data or structure location; in the second, *α* is a storage location.

Note that from the definitions of *yield* and ***val*** and the strictness of the basic situational functions, it follows that

- if s is the undefined state $\perp_s$ or if ***loc(s, x)*** is the undefined location $\perp_\ell$, then ***yield(s, x)*** is $\perp_\ell$

and

- if s is $\perp_s$, or if ***loc(s,*** x) is $\perp_\ell$, or if ***yield(s, x)*** is $\perp_\ell$, then ***val(s, x)*** is $\perp_d$.

From the definitions of the *yield* and data operators, we can prove the following ***value*** *property of identifiers:*

<3.11> ***val(s,*** x) = *data(s,* ***loc(s,*** *x)).*

Proof:

We distinguish between three cases:

Case: ***isdata(s, loc(s,*** *x)) or* ***isstructure(s, loc(s,*** x)), Then, ***yield(s, x)*** is defined to be ***loc(s, x)***. The property reduces to the definition of ***val.***

Case; ***isstore(s, loc(s,*** x)). Then,

val(s, *x)* = *data(s,* ***yield(s,*** *x))*
by the definition of *uaf* <3.10>

= data(s, ***store(s, loc(s,*** x)))
by the definition of ***yield*** <3.9>

= *data(s,* ***loc(s,*** *x))*
by the assumed property of data <3.7>

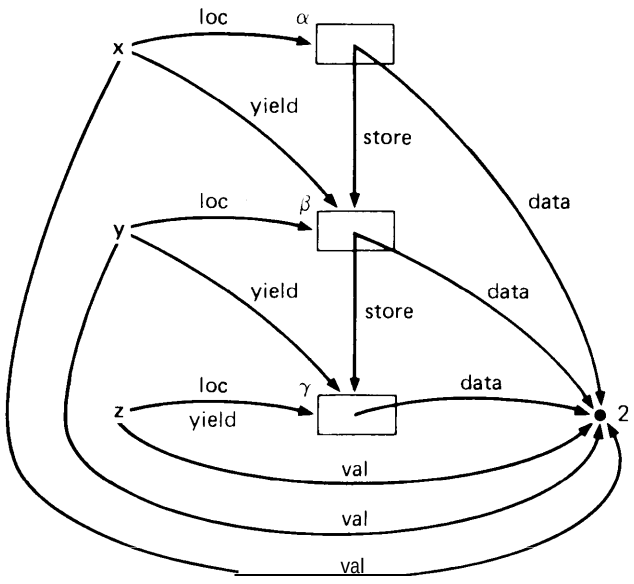
Case: ***loc(s,*** *x))* $=\perp_\ell$. Then ***val(s,*** x) $=\perp_d$ and ***data(s, loc(s, x))*** $=\perp_d$, by the strictness of the situational functions. ∎

In the future, we will often ignore the undefined case if it follows directly from the strictness of the situational functions.

Ezampfe (pointer):

Consider the configuration illustrated by the following diagram:

Figure 3.9



Here, α and β are storage locations and 7 is a data location. The three identifiers **x, y,** and z are bound to different locations, the evaluation of x and y yields different locations, but the values of x, **y,** and **z** are the same; i.e.,

$$val(s, x) = val(s, y) = val(s, z) = 2 \quad \blacksquare$$

In general, we shall say that a state s is numerically *faithful* if

$$val(s, 1) = 1$$

$$val(s, 2.2) = 2.2$$

$$val(s, -4) = -4$$

etc, for all numerical identifiers in *PL.* In some versions of FORTRAN one can construct defined states that are not numerically faithful. In such a (pathological) state, the value of the identifier **2** might be the data object 1.

D. Levels of Equality

In a situational-calculus framework, we can define four different relationship8 between two identifiers x and **y,** each of which causes them to have the same value.

- *identity*:

Here, x and y are identical: they stand for the same identifier; i.e.,

$$\mathbf{x} = \mathbf{y}.$$

Hence,

$$loc(s, x) = loc(s, y),$$
$$yield(s, x) = yield(s, y), \text{ and}$$
$$val(s, x) = val(s, y),$$

in any state s.

- *aliasing*:

Here, x and y may be syntactically distinct; i.e., possibly $x \neq y$. But x and y are *aliases* in the given state s: they are bound to the same location; i.e.,

$$loc(s, x) = loc(s, y)$$

and hence

$$yield(s, x) = yield(s, y), \text{ and}$$
$$val(s, x) = val(s, y).$$

- *equivalence*:

Here, x and y may be syntactically distinct, i.e., possibly $x \neq y$; and, in the given state s, x and y may not be aliases, i.e., possibly $loc(s, x) \neq loc(s, y)$; but x and y are *equivalent:* the evaluation of each of them yields the same location; i.e.,

$$yield(s, x) = yield(s, y),$$

and hence

$$val(s, x) = val(s, y).$$

(In LISP terminology, one would say that $eq(x, y)$ is true.)

• *equality:*

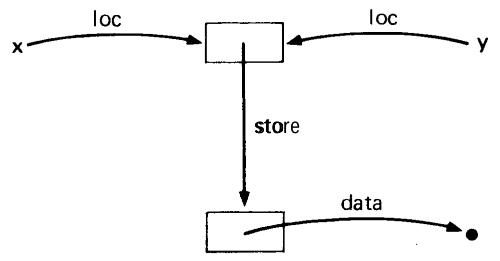
Here, x and y may be syntactically distinct, i.e., possibly $\mathbf{x} \neq \mathbf{y}$; in the given state s, x and y may not be aliases, i.e., possibly $loc(s, \mathbf{x}) \neq loc(s, \mathbf{y})$; and $\mathbf{x}$ and $\mathbf{y}$ may not be equivalent, i.e., possibly $yield(s, \mathbf{x}) \neq yield(s,$ y); but **x and** y are *equal,* in that they have the same value; i.e.,

$$val(s, x) = val(s, y).$$

(This is the standard notion of equality in most programming languages.)

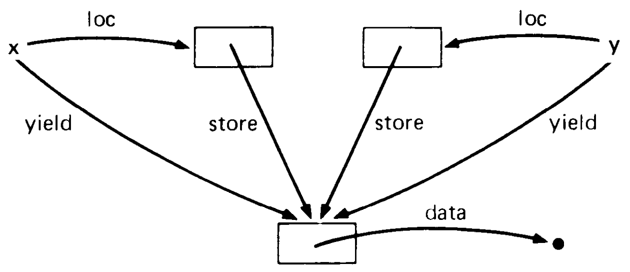
The notions of aliasing, equivalence, and equality can be illustrated (for the case in which $loc(s,$ x) and $loc(s, \mathbf{y})$ are storage locations) by the following diagrams:

Figure 3.10



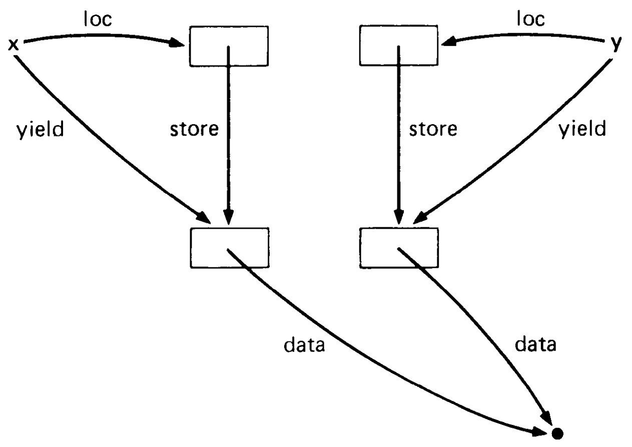
Here, x and y are aliases.

. Figure 3.11



Here, x and **y** are equivalent (but not aliases).

Figure 3.12



Here, x and y are equal (but not equivalent and not aliases).

E. Expressions

Up to this point, the only expressions of the programming language for which the situational operators have been defined are identifiers; we now extend the yield and *val* functions to other programming language expressions. For the time being, however, the function *loc* will be defined only for identifiers.

For any expression e in the programming language:

- *the yield of an expression*:

  yield(*s*, e) is the location yielded by the evaluation of e.

- *the value of an expression*:

  val(*s*, e) is the data object represented by *yield*(*s*, e) after the evaluation of e; thus

<3.12> $val(s, e) = data(s; e, yield(s, e))$.

In other words, to determine the value of e in state *s*, we first evaluate e, producing a new state s; e and yielding a location *yield*(*s*, e). We then determine the data object *data*(*s*; e, *yield*(*s*, e)) represented by this location in the new state.

In general, we assume that, if the evaluation of an expression yields a defined location, then the evaluation produces a defined state, and conversely; i.e.,

<3.13> $yield(s, e) = \perp_l$ if and only if $s; e = \perp_s$.

for all states s and expressions e in *PL*. This implies that, for any expressions $e_1, e_2, \ldots, e_n$ in *PL* and state *s*,

<3.14> if $yield(s; e_1; e_2; \ldots; e_{i-1}, e_i) = \perp_l$ for some i, $1 \leq i \leq n$
then $s; e_1; e_2; \ldots; e_n = \perp_s$.

For if

$$yield(s; e_1; e_2; \ldots; e_{i-1}, e_i) = \perp_l \quad \text{for some } i \text{ between 1 and } n$$

then (by <3.13>)

$$s; e_1; e_2; \ldots; e_{i-1}; e_i = \perp_s$$

and (by repeated application of <3.2>)

$$s; e_1; e_2; \ldots; e_n = \perp_s.$$

Informally, an expression e is said to have no side effects if its evaluation in a state s produces a state "indistinguishable" from s We will define two states s_1 and s_2 to be **indistinguishable,** denoted by $s_1 \approx s_2$, if the situational **operators** all behave the same in s_1 and s_2; i.e., if

<3.15> $loc(s_1, x) = loc(s_2, x)$,

<3.16> $store(s_1, a) = store(s_2, a)$,

<3.17> $data(s_1, a) = data(s_2, a)$,

<3.18> $yield(s_1, e) = yield(s_2, e)$,

<3.19> $isdata(s_1, a)$ if and only if $isdata(s_2, a)$,

<3.20> $isstore(s_1, a)$ if and only if $isstore(s_2, a)$,

<3.21> $isstructure(s_1, a)$ if and only if $isstructure(s_2, a)$

<3.22> $s_1 = \perp_s$ if and only if $s_2 = \perp_s$,

for all identifiers x, locations α, and expressions e. Clearly, $\approx$ is an equivalence relation.

We will say that an expression e *has* no *side effects* if its evaluation **is either** undefined or produces a state indistinguishable from the original state; i.e.,

$$\text{if } yield(s, e) \neq \perp_\ell \text{ then } s; e \approx s.$$

The evaluation of an identifier will be assumed to produce no side **effects;** **i.e.,**

<3.23> if $yield(s, x) \neq \perp_\ell$ then $s; x \approx s$,

for all states s and identifiers x in *PL*. (For the time being, we ignore the possibility that an identifier may denote a procedure, whose evaluation can produce side effects.) The definition of *val* for expressions e <3.12>, i.e.,

$$val(s, e) = data(s; e, yield(s, e))$$

can now be seen to be consistent with our earlier definition of *val* for identifiers x, *i.e.*,

$$val(s, x) = data(s, yield(s, x)).$$

For if $yield(s, x) \neq \perp_\ell$ then, by <3.23>, the evaluation of x produces no side effects; i.e., $s; x \approx s$, and hence, by <3.17>,

$$data(s; x, yield(s, x)) = data(s, yield(s, x)).$$

On the other hand, if $yield(s, x) = \perp_\ell$, then

$$data(s; x, yield(s, x)) = \perp_d = data(s, yield(s, x)).$$

In either case, the desired conclusion holds.

The only constructs we have introduced into the programming language ***PL*** so far are the identifiers and the sequencing operator ";". We will assume that ";" satisfies the following property .

- *the yield of* a *sequence:*

The location yielded by the evaluation of $e_1; e_2$ is precisely the location yielded by the evaluation of e_2 after the evaluation of e_1; i.e.,

<3.24> $$yield(s, e_1;e_2) = yield(s; e_1, e_2).$$

This implies the *value* of a *sequence* property

<3.25> $$val(s, e_1;e_2) = val(s; e_1, e_2).$$

For,

$$val(s, e_1;e_2) = data(s; (e_1;e_2), yield(s, e_1;e_2))$$

by the definition of
the value of an expression <3.12>

$= data((s;\ \mathbf{e}_1);\ \mathbf{e}_2,\ yield(s,\ \mathbf{e}_1;\mathbf{e}_2))$
by the sequence property <3.1>

$= data((s;\ \mathbf{e}_1);\ \mathbf{e}_2,\ yield(s;\ \mathbf{e}_1,\ \mathbf{e}_2))$
by the yield of a
sequence property above < 3.24 >

$= val(s;\ \mathbf{e}_1,\ \mathbf{e}_2)$
by the definition of the
value of an expression <3.12>

This concludes the introduction of our descriptive apparatus. In the following sections we will apply these notions to particular classes of expressions and simple programming languages.

4. STATIC EXPRESSIONS

In this section we describe a very simple class, the "static expressions," whose evaluation produces no side effects and involves no procedure calls. **This class is** not of great interest in itself, but the discussion will illustrate the application of our situational operators. Moreover, static expressions will be **a** component **of** other, more complicated expressions.

A. Pure Constructs

Certain constructs of a programming language, such as (typically) arithmetic operators, produce no side effects. We will say that a. construct $\mathbf{f}(\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_n)$ is *pure* if it has the following properties:

- *no aide effects*:

The evaluation of a pure construct, if it is defined, produces no side effects; i.e.,

<4.1> $\quad$ if $yield(s, \mathbf{f}(\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_n)) \neq \perp_\ell$
then $s; \mathbf{f}(\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_n) \approx s; \mathbf{e}_1; \mathbf{e}_2; \ldots; \mathbf{e}_n$

for all states s and expressions $\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_n$ of *PL.* Note that the operators $\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_n$ themselves may produce side effects, and that expressions **are** evaluated in left-to-right order. Special treatment is required for languages, such as ALGOL 68, in which the evaluations of $\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_n$ may be interleaved.

- *pure value:*

The construct **f** corresponds to a function $f_d(d_1, d_2, \ldots, d_n)$, mapping **data** objects into data objects, such that

<4.2> $\quad val(s, \mathbf{f}(\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_n)) =$
$f_d(val(s, \mathbf{e}_1), val(s; \mathbf{e}_1, \mathbf{e}_2), \ldots, val(s; \mathbf{e}_1; \mathbf{e}_2 \ldots; \mathbf{e}_{n-1}, \mathbf{e}_n))$

for all states s and expressions $\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_n$ in *PL.* We assume that f_d is *strict;* **i.e.,** if any of $d_1, d_2, \ldots, d_n$ is the undefined **data** object $\perp_d$ then $f_d(d_1, d_2, \ldots, d_n) = \perp_d$ *as* well.

For example, the arithmetic operators of algebraic languages and the **binary**-tree functions **left** and **right** are usually pure constructs; random number generators and the *cons* function of LISP produce side effects and are not pure.

B. Static Expressions

We now introduce static expressions, which consist of sequences of nested pure constructs and identifiers. More precisely, we define the static expressions by the following rules:

- every identifier x is a static expression.
- if f is a pure construct and $e_1, e_2, \ldots, e_n$ are static expressions then $f(e_1, e_2, \ldots, e_n)$ is a static expression.
- if $e_1, e_2, \ldots, e_n$ are static expressions, then $e_1; e_2; \ldots, e_n$ is a static expression.

For exampl'e, if **plus(u,** v) and **times(u,** v) are pure constructs, then

times(plus(x; 1, y), 2; x);1

is **a** static expression.

From this definition we can prove the following lemma.

Lemma (static expression):

The evaluation of **a** static expression e produces no side effects; i.e.,

<4.3> if $yield(s, e) \neq \perp_\ell$ then $s; e \approx s$,

for all states s.

The proof is by induction on the structure of the expression e; we assume inductively that the property holds for all proper subexpressions of e. Suppose that $yield(s, e) \neq \perp_\ell$. The proof distinguishes between several **cases,** depending on the structure of e.

Case: e is an identifier. Then $s; e \approx s$, because we have assumed **that** $yield(s, e) \neq \perp_\ell$ and that the evaluation of identifiers produces **no** side effects <3.23>.

Case: e is *of form* $\mathbf{f}(\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_n)$, *where* f *is a pure construct and* $\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_n$ are *static expressions.* We have assumed that

$$yield(s, \mathbf{f}(\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_n)) \neq \perp_\ell;$$

it follows, because pure constructs produce no side effects <4.1>, that

$$s; \mathbf{f}(\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_n) \approx s; \mathbf{e}_1; \mathbf{e}_2; \ldots; \mathbf{e}_n$$

and also (by <3.13>) that

$$s; \mathbf{f}(\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_n) \neq \perp_s.$$

Hence, by <3.22>,

$$s; \mathbf{e}_1; \mathbf{e}_2; \ldots; \mathbf{e}_n \neq \perp_s$$

and, by <3.14>,

$$yield(s; \mathbf{e}_1; \ldots; \mathbf{e}_{i-1}, \mathbf{e}_i) \neq \perp_\ell$$

for each i, $1 \leq i \leq n$.

Then

$$s; \mathbf{f}(\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_n) \approx s; \mathbf{e}_1; \ldots; \mathbf{e}_n$$
by <4.1> again

$$\vdots$$

$$\approx s; \mathbf{e}_1; \ldots; \mathbf{e}_i$$
by repeated application
of the induction hypothesis
(and the transitivity of $\approx$)

$$\vdots$$

$$\approx s.$$

Note that we were able to apply the induction hypothesis above only because we had established that

$$yield(s; \mathbf{e}_1; \ldots; \mathbf{e}_{i-1}, \mathbf{e}_i) \neq \perp_\ell$$

for each i.

Case: e is *of form* $e_1; e_2; \ldots e_n$, *where* $e_1, e_2, \ldots, e_n$ *ore static expressions.* We have assumed that

$$yield(s, e_1; e_2; \ldots; e_n) \neq \bot_t;$$

thus, by <3.13>,

$$s; e_1; \ldots; e_n \neq \bot_s.$$

It follows (by <3.14>) that

$$yield(s; e_1; e_2; \ldots; e_{i-1}, e_i) \neq \bot_t$$

for each i between 1 and n.

Then

$$s; (e_1; e_2; \ldots; e_n) \approx s; e_1; e_2; \ldots; e_n$$

by repeated application of the sequence property <3.1>

$\vdots$

$$\approx s; e_1; e_2; \ldots; e_i$$

by repeated application of the induction hypothesis

$\vdots$

$$\approx 8.$$

Again, we needed to establish that

$$yield(s; e_1; e_2; \ldots; e_{i-1}, e_i) \neq \bot_t$$

for each i, to apply the induction hypothesis. ∎

Corollary. If e is a statiс expression, then

<4.4> $val(s, e) = data(s, yield(s, e))$ for every state 8.

For, by the definition of the value of an expression,

$$val(s, e) = data(s; e, yield(s, e)).$$

In the case that $yield(s, e) \neq \perp_\ell$, the lemma tells us that $s; e \approx s$ and hence (by <3.17>)

$$data(s; e, yield(s, e)) = data(s, yield(s, e)).$$

On the other hand, if $yield(s, e) = \perp_\ell$, then

$$data(s; e, yield(s, e)) = \perp_d = data(s, yield(s, e))$$

Corollary. If $f(u_1, u_2, \ldots, u_n)$ is a pure construct and $e_1, e_2, \ldots, e_n$ are static expressions, then

<4.5> $$val(s, f(e_1, e_2, \ldots, e_n)) = f_d(val(s, e_1), val(s, e_2), \ldots, val(s, e_n)),$$

where f_d is the data object function corresponding to f.

Proof:

We have that

$$val(s, f(e_1, e_2, \ldots, e_n)) = \\ f_d(val(s, e_1), val(s; e_1, e_2), \ldots, val(s; e_1, e_2, \ldots; e_{n-1}, e_n)$$

by the pure-value property <4.2>. If we can show that

$$s; e_1; e_2; \ldots; e_i \approx s$$

for each i, $1 \leq i \leq n$, this will imply the desired result, by <3.17> and <3.18>.

In the case that

$$yield(s, f(e_1, e_2, \ldots, e_n)) \neq \perp_\ell,$$

we have (by <3.13>) that

$$s; f(e_1, e_2, \ldots, e_n) \neq \perp_s$$

and, hence, because pure constructs produce no side effects <4.1> and by <3.22>, that

$$s;e_1;e_2;\ldots;e_n \neq \perp_s.$$

By <3.14> then, this tells us that

$$yield(s;e_1;\ldots;e_{i-1},e_i) \neq \perp_\ell$$

for each i, $1 \leq i < n$. Finally, by repeated application of the static expression lemma itself, we conclude that

$$s;e_1;\ldots;e_{i-1};e_i \approx s$$

for each i, implying the desired result.

On the other hand, in the case that

$$yield(s, f(e_1, e_2, \ldots, e_n)) = \perp_\ell,$$

we would like to show that

$$f_d(val(s,e_1), val(s,e_2), \ldots, val(s,e_n)) = \perp_d.$$

If $val(s,e_i) = \perp_d$ for any i, the desired result follows from the strictness of f_d <4.2>; so we can assume $val(s,e_i) \neq \perp_d$ for each i. Consequently, $yield(s,e_i) \neq \perp_\ell$ for each i, by the definition of the val operator <3.10>. By repeated application of the lemma, then, we have that

$$\begin{gathered} s;e_1 \approx s \\ s;e_1;e_2 \approx s \\ \vdots \\ s;e_1;e_2;\ldots;e_i \approx s \end{gathered}$$

for each i, by the transitivity of $\approx$, implying the desired result. ∎

To prove the Hoare assignment axiom, the right-hand side of an assignment statement will be restricted to be a static expression. But first we must introduce the assignment statement itself.

5. THE ASSIGNMENT STATEMENT

We are now ready to define the semantics of a simple assignment statement in terms of the situational operators. The statement we describe is only a prototype: different programming languages provide different varieties of assignment. For these languages, our definition can be altered accordingly.

An *assignment statement* in *PL* is of form

$$\mathbf{x} \leftarrow \mathbf{e},$$

where x is an identifier and e is any expression in *PL*. The execution of this statement in a state s takes place in three stages:

(1) The location $loc(s, \mathbf{x})$ associated with the identifier x is determined. It is assumed that this step produces no side effects. If $loc(s, \mathbf{x})$ is not a storage location, an "error condition" occurs; in this case, we will say that

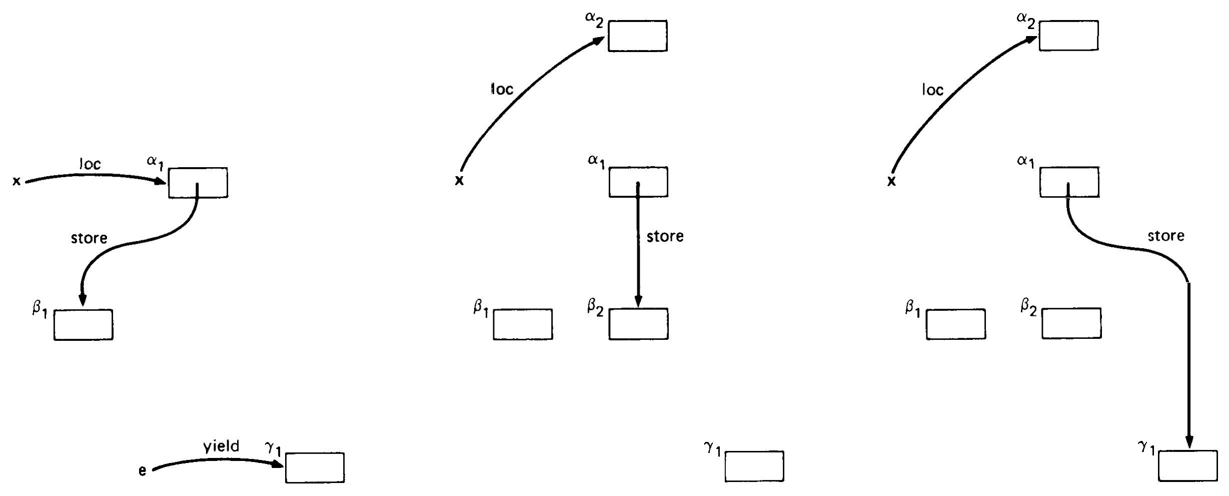
$$s; \mathbf{x} \leftarrow \mathbf{e} = \perp_s.$$

(2) The expression e is evaluated, yielding the location $yield(s, \mathbf{e})$ that represents the value of e, and producing a new state $s; \mathbf{e}$. (In general, the evaluation of e may produce side effects.)

(3) The location $yield(s,$ e) is stored in the location $loc(s, \mathbf{x})$, producing the new state s; x $\leftarrow$ e. The location yielded by the evaluation of the assignment statement itself is the same as that yielded by the evaluation of e.

We illustrate this process diagrammatically as follows:

Figure 5.1



at *8* at $s;e$

Note that in the second picture, **correponding** to the state $s;e$, the *loc* link leading from x and the *store* link leading from α_1 have been changed; this is because **the** evaluation of e may have side effects that can change any of the links at state s.

The information suggested by the above diagram is conveyed more precisely, if perhaps less readably, by the following *assignment axioms*. These express **the** effect of the assignment statement on **each** of the situational operators:

- *principal axiom*

<5.1> $store(s;\mathbf{x} \leftarrow \mathbf{e}, loc(s,\mathbf{x})) = yield(s,\mathbf{e})$ if $isstore(s, loc(s,\mathbf{x}))$

for all states s and identifiers x and expressions **e** in *PL*. **The above axiom** describes the change that is the intended effect of the assignment statement. **It** is also necessary to introduce *frame axioms* indicating that **no** other **changes are** produced by the assignment.

- *frame axioms*

<5.2> $loc(s;\mathbf{x} \leftarrow \mathbf{e}, y) = loc(s;\mathbf{e}, y)$

<5.3> $\mathit{store}(s; x \leftarrow e, \ell) = \mathit{store}(s; e, \ell)$ if $\ell \neq \mathit{loc}(s, x)$

<5.4> $\mathit{data}(s; x \leftarrow e, \ell) = \mathit{data}(s; e, \ell)$ if $\mathit{isdata}(s; e, \ell)$

<5.5> $\mathit{isdata}(s; x \leftarrow e, \ell)$ if and *only* if $\mathit{isdata}(s; e, \ell)$

<5.6> $\mathit{isstore}(s; x \leftarrow e, \ell)$ if and only if $\mathit{isstore}(s; e, \ell)$

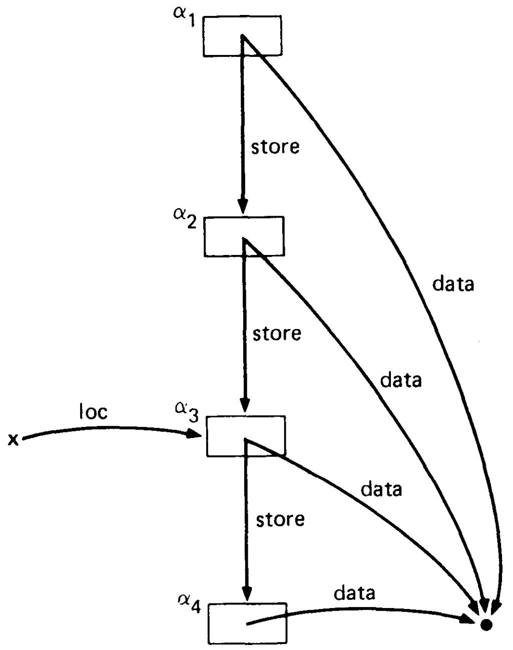
<5.7> $\mathit{isstructure}(s; x \leftarrow e, \ell)$ if and only if $\mathit{isstructure}(s; e, \ell)$

for all states s, identifiers **x** and **y**, expressions **e, and locations ℓ, such that**

$$\mathit{isstore}(s, \mathit{loc}(s, x)).$$

The axioms express that only the store link of $\mathit{loc}(s, x)$ can be altered by the execution of the assignment itself. Note that other links can be **altered by the** evaluation of e. **Also** note that, although the assignment is assumed to leave **the** *data* link of data locations unchanged, it can indirectly alter the data link of **other** locations. For example, consider the configuration illustrated below:

Figure 5.2



Here, evaluating an assignment statement $x \leftarrow$ e will alter the ***store*** link **of** α_3. Consequently, it will indirectly alter the *data* link of the storage locations α_3, α_2, and α_1, by virtue of the relationship <3.7> we have assumed, ***viz.,***

$$\mathit{data}(s, \ell) = \mathit{data}(s, \mathit{store}(s, \ell)) \text{ if } \mathit{isstore}(s, \text{C}).$$

Structure locations can also have their data links altered indirectly by assignment statements, as will be seen in the forthcoming Part II.

Because we regard assignment statements as expressions which may appear as subexpressions of other expressions (e.g., we consider $2 + (x \leftarrow (x - 1))$ to be a legal expression), we need an axiom that defines the location representing the value of an assignment statement.

- *yield* axiom:

<5.8> $yield(s, x \leftarrow e) = yield(s, e)$ if $isstore(s, loc(s, x))$

for all states s, identifiers **x**, and expressions e. In other words, the location yielded by the evaluation of the assignment statement itself is the same as that yielded by the evaluation of **e**.

- *illegal axiom*:

<5.9> $s; x \leftarrow e = \perp_s$ if *not' isstore(s, loc(s, x))*.

- *undefined axiom:*

<5.10> $s; x \leftarrow e = \perp_s$ if $s; e = \perp_s$.

In other words, if **x** is not associated with a storage location when the assignment is executed, an error condition is produced; and if the evaluation of e in state *s* is undefined, then the evaluation of the assignment statement is also undefined.

Let us illustrate the use of these axioms to prove a simple property that is beyond the expressive power of the Hoare rules. We show that if e is an expression that creates no aliasing, then $x \leftarrow e$ also creates no aliasing; i.e.,

Lemma (no aliasing):

Suppose e is an expression in *PL* with the property that

(*) if $loc(s, x) \neq loc(s, y)$ then $loc(s; e, x) \neq loc(s; e, y)$

for all states s and identifiers **x** and y. Then

$$\text{if } loc(s, \mathbf{x}) \neq loc(s, y) \text{ then } loc(s; \mathbf{z} \leftarrow e, x) \neq loc(s; \mathbf{z} \leftarrow e, y),$$

for all states s and identifiers **x**, y, and **z**, such that

$$isstore(s, loc(s, \mathbf{z})).$$

Proof:

Assume that e has the above property (*) and that

$$loc(s, x) \neq loc(s, y).$$

Then

$loc(s; z \leftarrow e, \mathbf{x}) = loc(s; e, x)$
by the frame axiom
for assignment < 5.2 >
$\neq loc(s; e, y)$
by the assumed property (*)
$= loc(s; z \leftarrow e, y)$
by the frame axiom for assignment, again.

In short,

$$loc(s; z \leftarrow \mathbf{e}, \mathbf{x}) \neq loc(s; z \leftarrow e, y). \quad \blacksquare$$

6. ASSERTIONS WITHIN STATES

Our situational operators can describe properties of a program in terms of machine locations. Often it is necessary to express such properties in terms of *assertions,* where an assertion is a relationship among the values of the program's identifiers that is intended to hold when the execution reaches a certain point in the program. In this section we connect these two levels of description, assertions and situational **operators**.

A. Assertion Language

An assertion will customarily involve constructs from both the programming language and the program's subject domain. For example, suppose we have a program that is intended to assign to an identifier z the greatest common divisor *gcd(x, y)* of (the values of) two identifiers **x** and y, without changing **x** and y. Then we might wish to assert that, when the execution of the program terminates, the following relationship is true:

$$\mathbf{z} = max\{u : u|\mathbf{x} \text{ and } u|\mathbf{y}\}.$$

Here **x,** y, and z are identifiers in the programming language; the other symbols *maz,* *{u: . . .},* |, and and are constructs from the theory of the program's subject domain.

In general, then, we assume we have a *domain language DL* for expressing sentences about the program's subject domain; we would like to extend this language to form an assertion *language* AL, that also includes identifiers and pure constructs from the programming language *PL.* This extension can be achieved with no confusion, because we have adopted disjoint vocabularies for the domain language and the programming language. In our discussion, subject domain constructs will always be denoted by italic *(italic)* characters while programming language constructs will be denoted by boldface (**boldface**) characters.

The truth of an assertion in *AL* is only meaningful with respect to a state s of the execution; each programming language identifier **x** in the assertion is then taken to refer to the data object $val(s, \mathbf{x})$ in the program's domain. For example, the above assertion

$$\mathbf{z} = max\{u : u|\mathbf{x} \text{ and } u|\mathbf{y}\}$$

is true in a state s if the sentence

$$val(s, z) = max\{u : u|val(s, x) \text{ and } u|val(s, y)\}$$

is true in the subject domain.

B. The Extended *val* Operator

To discuss the truth of a sentence in the assertion language, we allow assertion-language expressions among the objects'we consider, and extend the situational operator *val* to apply to such expressions. We will then say that an assertion P holds in a state *s* if $val(s, P) = true$.

The extension of the *val* operation is similar to Tarski's definition of the truth of sentences in logic. We will pay special attention to quantified expressions in the assertion language. We want to say that an expression $\forall uQ$ has value *true* in a state s if Q bas value true in s regardless of the value of the dummy variable u. To formalize thrs definition, we introduce the notion of a *substantiation* ψ, a function that maps some of the variables of the domain language into data objects in D. We will add a substantiation as a third argument to the *val* operator, $val(s, t, \psi)$; our intention is that the substantiation ψ will supply the values of any free variables in the assertion language expression t. We will then be able to define the *val* of $\forall uQ$, say, in terms of the *val* of Q, for appropriate substantiations.

Let us be more precise. If $u_1, u_2, \ldots, u_k$ are variables in *DL* and $d_1, d_2, \ldots, d_k$ are data objects, then the list

$$\psi: \quad \langle u_1 \leftarrow d_1, u_2 \leftarrow d_2, \ldots, u_k \leftarrow d_k \rangle$$

is a substantiation. We do not assume that the variables $u_1, u_2, \ldots, u_k$ are distinct, and we do regard the order of the list as significant. We will say that the variables $u_1, u_2, \ldots, u_k$ are substantiated by ψ. We denote by $\langle\,\rangle$ the *empty substantiation,* which substantiates no variables. If u is a variable, d is a data object, and ψ is as above, we denote by $(u \leftarrow d) \circ \psi$ the *eztended substantiation*

$$\langle u \leftarrow d, u_1 \leftarrow d_1, u_2 \leftarrow d_2, \ldots, u_k \leftarrow d_k \rangle.$$

Henceforth, we will regard *val* as a situational function $val(s, t, \psi)$ of three arguments: a state s, an expression t of AL, and a substantiation ψ. We consider the earlier notation $val(s,t)$ with two arguments as an abbreviation for $val(s,t,\langle\rangle)$, where ( ) is the empty substantiation; i.e.,

<6.1> $val(s,t) = val(s,t,(\ ))$.

We define the language AL and the extended *val* function **according to the** following rules:

- As usual, in the undefined state $\perp_s$,

<6.2> $val(\perp_s, t, \psi) = \perp_d$

for any expression t in AL and any substantiation ψ

- A constant c in *DL* is a constant of AL and is associated with a data object c_d in D. Thus,

<6.3> $val(s, c, \psi) = c_d$

for all (defined) states s and all substantiations ψ.

- All variables of *DL* are variables of AL. For distinct variables u and v in *DL,*

<6.4> $val(s, u, \langle\ \rangle) = \perp_d$

<6.5> $val(s, u, (u \leftarrow d) \circ \psi) = d$

<6.6> $val(s, u, \langle v \leftarrow d) \circ \psi) = val(s, u, \psi)$

for all (defined) states s, data objects d and substantiations ψ. Note that if a variable occurs more than once in a substantiation, the leftmost occurrence predominates; e.g.,

$$val(s, u, (u \leftarrow d_1, u \leftarrow d_2, u \leftarrow d_3)) = d_1.$$

On the other hand, if a variable in *DL* is not substantiated at all, its value **is** undefined; thus if u is distinct from variables v and w, then

$$val(s, u, \langle v \leftarrow d_1, w \leftarrow d_2\rangle) = \perp_d.$$

- If x denotes an identifier in *PL,* then x is in *AL* and

<6.7> $val(s, x, \psi) = val(s, x)$

for all states s and substantiations ψ. Thus, the value of an identifier is independent of the substantiation ψ.

• A function symbol g in *DL* is a function symbol of **AL and is associated with a** strict function g_d over the set *D* of objects; thus

<6.8> $$val(s, g(t_1, t_2, \ldots, t_n), \psi) = g_d(val(s, t_1, \psi), val(s, t_2, \psi), \ldots, val(s, t_n, \psi))$$

for all states s, expressions $t_1, t_2, \ldots, t_n$ in AL, and substantiations ψ.

Similarly, a predicate symbol in *DL* is associated with a strict function over the set *D* of objects; a similar definition applies.

• If **f** denotes a pure construct in *PL* , then f is a function symbol in AL and is associated with a strict function f_d over the set *D* of data objects (see **<4.2>);** then

<6.9> $$val(s, \mathbf{f}(t_1, t_2, \ldots t_n), \psi) = f_d(val(s, t_1, \psi), val(s, t_2, \psi), \ldots, val(s, t_n, \psi))$$

for all states s, expressions $t_1, t_2, \ldots, t_n$ in AL, and substantiations ψ.

• The value of a logical expression of form $\neg P$ in AL is the negation of the value of P; i.e.,

<6.10> $$val(s, \neg P, \psi) = not(val(s, P, \psi))$$

for all states s and substantiations ψ.

• The value of a logical expression of form $P \wedge Q$ in AL is the conjunction of **the** values of *P* and Q; i.e.,

<6.11> $$val(s, P \wedge Q, \psi) = val(s, P, \psi) \text{ and } val(s, Q, \psi).$$

The other logical connectives are treated similarly.

• The value of a quantified expression $\forall x P$ in AL (where x is **a** variable in *DL* and *P* is *a* logical expression in *AL)* is true if the value of *P* is true regardless of how we substantiate x; more precisely:

<6.12> $$val(s, \forall x P, \psi) = true$$

if, for every data object d, $val(s, P, (x \leftarrow d) \circ \psi) = true$

<6.13> $val(s, \forall x P, \psi) = julse$
if, for some data object d, $val(s, P, (x \leftarrow d) \circ \psi) = false$

<6.14> $val(s, \forall x P, \psi) = \perp_d$ otherwise

Note that the value of $\forall x P$ will be undefined if P is undefined for some substantiations but P is never false.

Because $\exists x P$ is logically equivalent to $\neg \forall x \neg P$, the above definition suggests a definition of $val(s, \exists x P, \psi)$; i.e.,

<6.15> $val(s, \exists x P, \psi) = val(s, \neg \forall x \neg P, \psi)$.

• The value of the set constructor {u: P} in AL is a set of all data objects in D that satisfy P; more precisely:

<6.16> $val(s,$ {u: $P\}, \psi)$ = the set of all data objects d in D such that $val(s, P, (u \leftarrow d) \circ \psi) = true$

If the domain language includes other constructs; then val must be extended for them as well. Note that we do not include the "impure" constructs of the' programming language PL, i.e., those constructs with side effects, in the assertion language AL.

We will need the following simple properties of the extended val operator.

• The value of an expression ♦ in DL (which contains no identifiers of PL) is independent of the state; i.e.,

<6.17> $val(s,$ ♦📪 $\psi) = val(s',$ ♦📪 $\psi)$

for all states s and s' and substantiations ψ.

• The value of a static expression e in PL is independent of the substantiation; *i.e.*,

<6.18> $val(s, e, \psi) = val(s, e)$

for all states s and substantiations ψ. These properties can be proved from the definition of the extended val operator by induction on the structure of the substantiations and expressions.

C. The □ and ◇ Operators

Now that we have extended *val* to apply to expressions in the assertion language, we can define two situational operators, □ and 0, for expressing that an assertion P holds in a state s:

$$\langle 6.19\rangle \qquad \square \ (8, P) = \begin{cases} \text{true if } val(s, P) = true \text{ or } val(s, P) = \perp_d \\ \text{false if } val(s, P) = false \end{cases}$$

In short, $\square(s, P)$ holds if P is not false in a.

$$\langle 6.20\rangle \qquad \diamond(s, P) = \begin{cases} \text{true if } val(s, P) = true \\ \text{false if } val(s, P) = \text{false or } val(s, P) = \perp_d \end{cases}$$

In short, $\diamond(s, P)$ holds if P is true in s. Note that □ and o are always true or false; they are never undefined even if their arguments are. Furthermore, they satisfy the usual dualities of modal logic:

$$\langle 6.21\rangle \qquad \square(s, \neg P) = \neg\ o(s,P)$$

$$\langle 6.22\rangle \qquad \diamond(s, \neg P) = \neg \square(s, P).$$

These operators are distinct from the $[e]P$ and $< e > P$ of dynamic logic (Pratt [1976]) in that they are applied to states s rather than program segments **e**.

Typically, the □ operator is used to express the partial correctness of a program, the o operator to express its total correctness. For example, to represent a statement in Hoare's logic of form

$$\{P\}\ \mathbf{e}\ \{Q\},$$

where e is a program segment, we write

$$\text{if } \square \ (S, P) \text{ then } \square \ (S; e, Q) \text{ for all states } s.$$

This sentence expresses the partial correctness of the program segment e with respect to the input assertion P and the output assertion Q; it is automatically true if the program segment e fails to terminate; *i.e.*, if $s; \mathbf{e} = \perp_s$, because then

$$val(s; \mathbf{e},\ \ Q) = \perp_d$$

and, therefore,

$$\Box(s;e, Q) = true,$$

by the definition of $\Box$, <6.19>.

On the other hand, to express the *total correctness* of e with respect to the same assertions *P* and Q, we write

$$\text{if } \Diamond(s, P) \text{ then } \Diamond(s;e, Q) \text{ for } \mathbf{all} \text{ states } s.$$

7. PROVING HOARE'S ASSIGNMENT AXIOM

We have now introduced the concepts **necessary** for expressing the classical **Hoare** assignment axiom and the restrictions under which it is true. One virtue of this situational-calculus approach is that it allows us to formulate **these** restrictions explicitly in a mathematical language and to deal with them in the same framework in which we conduct all our reasoning. In this section, we will prove **the** "assignment theorem," that the **Hoare** axiom is true under the appropriate restrictions.

Usually the axiom is expressed **as**

$$\{P \lhd \langle \mathbf{x} \leftarrow \mathbf{e} \rangle\}\ \mathbf{x} \leftarrow \mathbf{e}\ \{P\},$$

where P is an assertion, x is an identifier, and e is an expression in $\boldsymbol{PL}$. Recall that $P \lhd \langle x \leftarrow e \rangle$ stands for the result of replacing all free occurrences of x in P by e. The axiom indicates that if the assertion $P \lhd \langle \mathbf{x} \leftarrow e \rangle$ holds before executing, the assignment $x \leftarrow e$, and if the execution terminates, then the assertion $\boldsymbol{P}$ holds afterwards.

Expressed in terms of the situational operator $\square$, the axiom reads

$$\langle 7.1 \rangle \qquad \begin{array}{l} \text{if } \square(s, P \lhd \langle \mathbf{x} \leftarrow e \rangle) \\ \text{then } \square(s;x \leftarrow e, P) \end{array}$$

for all states s, where $\square(s, Q)$ means that the assertion Q is either true in s or undefined in s.

A. Problematic Phenomena

In the section on MOTIVATION, we have given many situations arising in actual programming languages for which the above axiom is false. In this section we will review those phenomena, and specify the restrictions for precluding them.

- *the expression e must be static:*

This restriction prohibits assignment statements such as

$$x \leftarrow (y \leftarrow z).$$

Such statements violate the assignment axiom. For example, suppose "=" is the domain language equality predicate. Then, for a state s in which y = 1, it will be true that

$$\square(s, (y = 1) \triangleleft \langle x \leftarrow (y + 2) \rangle),$$

i.e.,

$$\square(s, y = 1).$$

However, after executing the assignment, y will be 2; i.e.,

$$\square(s; x \leftarrow (y + 2), y = 2);$$

so it will be false that

$$\square(s; x \leftarrow (y + 2), y = 1),$$

contradicting the axiom.

- *P must be an assertion in the assertion language AL:*

  Otherwise, P might be a condition such as

  "there exists an identifier whose value is 2."

Here P contains no occurrences of x at all. Therefore, if s is a state in which x is the only identifier whose value is 2, it is true that

$$\square(s, P \triangleleft \langle x \leftarrow 0 \rangle),$$

i.e.,

$$\square(s, P).$$

However, after executing the assignment x ← 0, there is no identifier whose value is 2; therefore, it is false that

$$\square(s; x \leftarrow 0, P),$$

contradicting the axiom.

- *x must not be an alias of any other identifier that* ***occurs*** *in P:*

For, suppose x and y are aliases, and s is a state in which y = 0. Then **if** we execute the assignment x ← 1, y will be changed to 1. **In** other words, **it is** true that

$$\Box(s, (y = 0) \lhd \langle x \leftarrow 1 \rangle),$$

i. e.,

$$\Box(s, Y = 0),$$

but false that

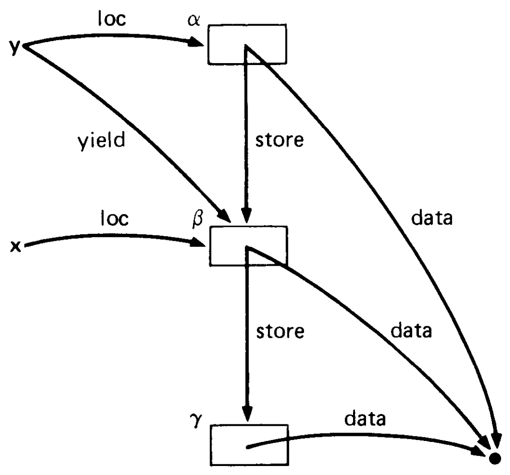
$$\Box \quad (8;x \leftarrow 1, y = o),$$

contradicting the axiom.

- *x must not be "pointed to" by any* ***expression***:

For, suppose a configuration such as the following exists **in state s;**

Figure 7.1



In other words, the location β to which the identifier **x** is bound **is** the **location** yielded by the evaluation of the identifier y. Then x and y both have the **same** value, 0 in this case, because the *store* operator transmits the **value (<3.7> and <3.11>)**. If we execute the assignment x ← 1, we indirectly alter the value **of y, to** 1. In other words, it is true that

$$\Box(s, (y = 0) \lhd \langle x \leftarrow 1 \rangle),$$

i. e.,

❑ ✡ 🖫 0),

but false that

$$\square(s; \mathbf{x} \leftarrow 1, \mathbf{y} = 0),$$

contradicting the axiom. Thus, problems may arise if some identifier y in the assertion *P* "points to" x.

Similar problems arise if x is "pointed to" by some identifier in the expression e. For example, suppose that in state s above, instead of executing x $\leftarrow$ 1 we execute x $\leftarrow$ y. Then the configuration in state s; x $\leftarrow$ y is

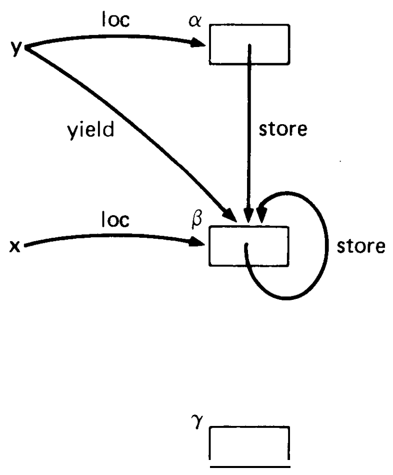


Figure *7.2*

Here, x is bound to a circular tree; although **we** have not yet **defined the data** object represented by such a structure, it is certain that this object **is not 0. Thus,** although it is true initially that

❑ (8, (X $\doteq$ 0) $\triangleleft$ $\langle \mathbf{x} \leftarrow \mathbf{y} \rangle$),

i. e.,

❑ ♒ ✡ 🖫 0),

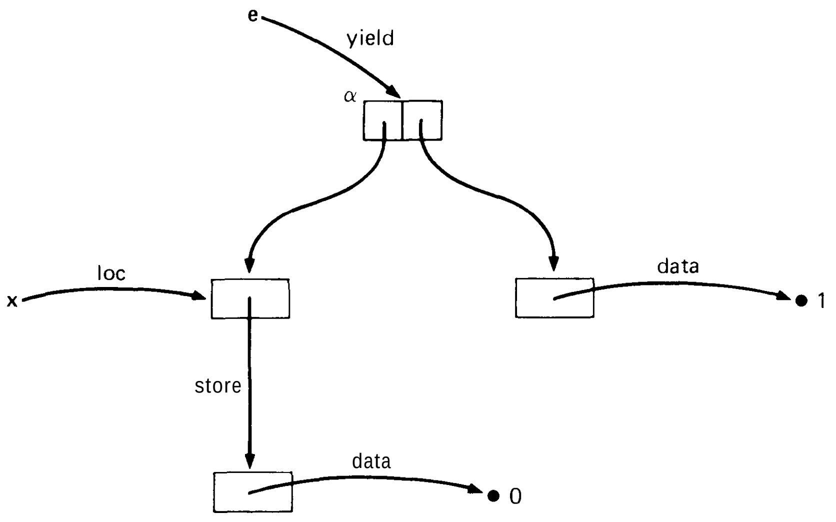
it is not true that

$$\square(s; \mathbf{x} \leftarrow \mathbf{y}, \mathbf{x} = \mathbf{o}),$$

thereby contradicting the axiom. Here, a problem occurs because x was pointed to by an identifier in e, not in *P*.

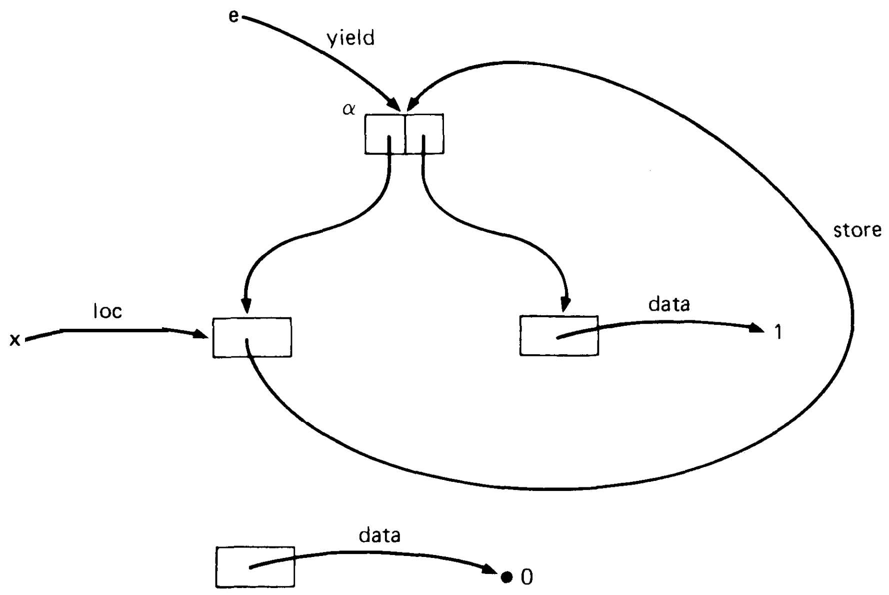
Finally, problems can occur if x is "pointed to" by an expression other **than** an identifier. For example, suppose that the following configuration occurs in state *s*:

Figure *7.3*



Here x is an identifier (whose value is 0), e is any static expression in *PL* that contains no occurrence of x, and *α* is a structure location whose left descendant is *loc*(*s*, x). The evaluation of the *PL* expression **left(e)** yields the location *loc*(*s*, x). Thus, in this state, **left(e)** = 0. If we then execute the assignment **x ← e**, we create the **following** configuration in *s*; x ← e

Figure *7.4*



Note that in this configuration **left(e)** $\neq$ 0. Hence, while it is true that

$$\Box \quad (s, (\text{left}(e) = 0) \lhd \langle x \leftarrow e \rangle),$$

i.e., (because x does not occur in e),

$$\Box(s, \text{left}(e) = 0),$$

it does not follow that

$$\Box(s; x \leftarrow e, \text{left}(e) = 0),$$

contradicting the axiom.

Surprisingly enough, no damage is done if x is the alias of some other identifier in e. For example, suppose that **x** and **y** are aliases, and *s* is a state in which x (and y) are 0. Then, if we execute the assignment x ← **y+1**, x (and y) will both be changed to 1. In other words, it is true that

$$\Box(s, (x = 1) \lhd (x \leftarrow y+1)),$$

i.e.,

$$\Box(s, y+1 = 1),$$

but it is also true that

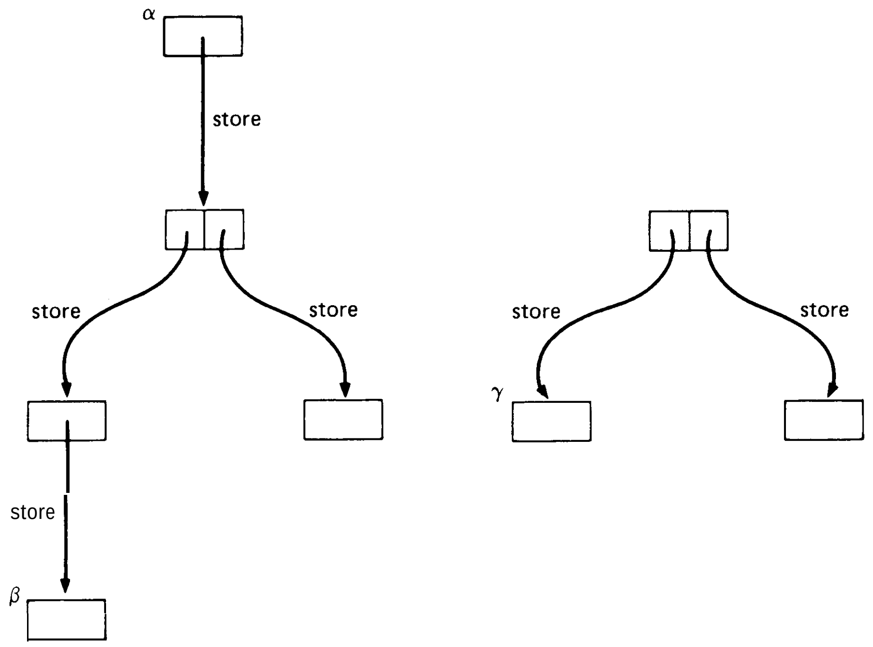
$$\Box(s; x \leftarrow y+1, x = 1),$$

as the axiom requires.

B. Accessibility

The informal notion of "pointing to" that we used casually in the last section, will be formalized by introducing a concept of "accessibility" in the forthcoming Part II of this paper, on data structures. Roughly speaking, we will say that a location β is *accessible* from a location α (in a given state) if there is some way of reaching β from a by applying a possibly empty sequence of *store* operators or other data structure operators. For example, consider the following configuration:

Figure *7.5*



Here, *β* is accessible from a, but *γ* is not.

Some more examples: if *β* = *store*(*s*, a) in a state *s*, then *β* is accessible from *α* in *s*; and if *β* and *α* are identical, then *β* is accessible from *α*. Furthermore, if two identifiers w and x are aliases, then *loc*(*s*, x) is accessible from *loc*(*s*, w).

An expression e in *PL* will be said to *point to* a location a (in a state *s*) if a is accessible (in s) from the location *yield*(*s*, e) yielded by the evaluation of e. On the other hand, a location will be said to be isolated (in *s*) if it is not pointed to by any expression in *PL*. Note that it is possible for an identifier to have aliases but still be bound to an isolated location; e.g., we may have *loc*(*s*, x) = *loc*(*s*, y), for two distinct identifiers x and y, where *loc*(*s*, x) is a storage location not accessible from any location other than itself.

Although we do not actually define accessibility formally until the forthcoming Part II, we can state the one "accessibility property" that we will need to prove the assignment theorem. Then, in Part II, we will prove this property as a lemma.

First, let *us* define a *static* ***assignment*** to be any assignment x ← e in which e is a static expression.

Now, to illustrate the accessibility property assume that the following configuration exists in state *s*:

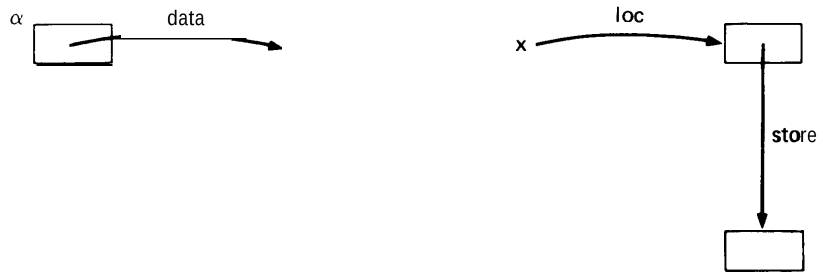


Figure 7.6

Here, α is any location and $loc(s, \mathbf{x})$ is a storage location. Then, if $loc(s, \mathbf{x})$ is not accessible from α, a static assignment x ← e cannot alter the data object represented by α.

In terms of our situational operators, this is expressed as follows.

Static accessibility property: Suppose s is a state, x is an identifier, and e is a static expression in *PL.* If $loc(s$, x) is not accessible from some location α in state 8, then

<7.2> $data(s; x \leftarrow e, a) = data(s, a).$

This is actually a special case of the following more general statement, which applies to nonstatic expressions e as well.

Accessibility property: Suppose s is a state, x an identifier, and e is any expression in *PL.* If $loc(s, \mathbf{x})$ is not accessible from some location α in state s; **e**, then

<7.3> $data(s; x \leftarrow e, a) = data(s;$ e, a), and

<7.4> $loc(s$, x) is not accessible from α in **state s; x ← e.**

C. Static Assignment Theorem

We are now is **a** position to state the restricted **Hoare** axiom as a theorem.

Theorem (Static *Assignment):* Let s be a state, x an identifier, e a static expression in PL, and P an assertion in AL, such that

(a) x is not an alias of any other identifier that occurs in P; i.e., $loc(s, \mathrm{x}) \neq loc(s, \mathrm{w})$ for any identifier w distinct from x that occurs in P.

(b) $loc(s, \mathbf{x})$ is isolated.

Then we have

<7.5> $\quad$ if $\square(s, P \lhd \langle \mathbf{x} \leftarrow \mathrm{e} \rangle)$
then $\square(s; \mathrm{x} \leftarrow \mathrm{e}, P)$,

The restriction (b) above is stronger than necessary, but it eliminates with *a* single stroke all the problematic phenomena we discussed involving expressions pointing to $loc(s, \mathrm{x})$.

The proof of the theorem relies directly on the following lemma.

Lemma (static assignment): Let s be a state, x an identifier, e a static expression in PL, t an expression in AL, and ψ a substantiation such that

(a) x is not an alias of any other identifier that occurs in t,

(b) $loc(s, \mathrm{x})$ is isolated.

Then we have

<7.6> $\quad$ if $val(s; \mathbf{x} \leftarrow \mathbf{e}, t, \psi) \neq \perp_d$
then $val(s; \mathrm{x} \leftarrow \mathrm{e}, t, \psi) = val(s, t \lhd \langle \mathbf{x} \leftarrow e \rangle, \psi)$.

Intuitively, the lemma asserts that, under suitable restrictions, the value of the expression t after executing the assignment will be the same as the value of t' before, where t' is the expression obtained from t by replacing every free occurrence of x by e. The third substantiation argument ψ is included to allow for the possibility that t contains free occurrences of variables from the domain language DL. The condition $val(s; \mathrm{x} \leftarrow \mathrm{e}, t, \psi) \neq \perp_d$ avoids the situation in which e contains some identifiers whose value is undefined in s; i.e., $val(s, \mathbf{e}) = \perp_d$; **for** then $s; \mathrm{x} \leftarrow \mathrm{e} = \perp_s$. Otherwise, if the identifier x does not occur in t and t **is** defined in s, it could happen that

$$val(s, t \lhd \langle \mathbf{x} \leftarrow \mathbf{e} \rangle, \psi) = val(s, t, \psi) \neq \perp_d,$$

because x does not occur in t, but

$$val(s; x \leftarrow e, t, \psi) = val(\perp_s, t, \psi) = \perp_d.$$

Lemmu $\Rightarrow$ *Theorem:* To see that the lemma implies the theorem, assume s, x, e, and P satisfy the restrictions of the theorem, and that

$$\Box(s, P \lhd \langle x \leftarrow e \rangle)$$

is true; i.e., either

$$val(s, P \lhd \langle x \leftarrow e \rangle, \langle\,\rangle) = true$$

or

$$val(s, P \lhd \langle x \leftarrow e \rangle, \langle\,\rangle) = \perp_d$$

by the definition of the $\Box$ operator <6.19> and the extension of *val* to **the** assertion language <6.1>.

We distinguish between two cases:

Case: $val(s;\ x \leftarrow e, P, \langle\,\rangle) = \perp_d$. Then the desired conclusion

$$\Box(s; x \leftarrow e, P)$$

is true, by <6.1> and <6.19>, again.

Case: $val(s; x \leftarrow e, P, \langle\,\rangle) \neq \perp_d$. Then by the lemma (taking t to be P),

$$val(s; x \leftarrow e, P, \langle\,\rangle) = val(s, P \lhd \langle x \leftarrow e \rangle, \langle\,\rangle).$$

But we know that

$$val(s, P \lhd \langle x \leftarrow e \rangle, \langle\,\rangle) = true \text{ or}$$
$$val(s, P \lhd \langle x \leftarrow e \rangle, \langle\,\rangle) = \perp_d,$$

and therefore

$$val(s; x \leftarrow e, P, \langle\,\rangle) = true \text{ or}$$
$$val(s; x \leftarrow e, P, \langle\,\rangle) = \perp_d.$$

In either case, the desired conclusion

$$\Box(s; x \leftarrow e, P)$$

is true, again by <6.1> and <6.19>. ∎

It remains to prove the lemma. The proof depends on the following proposition.

Proposition (static assignment): Suppose s is a state, x is an identifier, and e is a static expression in *PL,* such that $loc(s, \mathbf{x})$ is *a* storage location, i.e.,

$$isstore(s, loc(s, x)).$$

and that the execution of the assignment $x \leftarrow e$ is defined, i.e.,

$$s; \mathbf{x} \leftarrow \mathbf{e} \neq \perp_s.$$

Then

- The location bound to any identifier is unchanged by a static assignment; *i.e.,*

<7.7> $loc(s; x \leftarrow e, y) = loc(s, y)$ for all identifiers y

- Whether a location is a storage location is unaffected by a static assignment; i.e.,

<7.8> $isstore(s; x \leftarrow e, a)$ if and only if $isstore(s, a)$ for all locations a.

- The location yielded by evaluating x after executing the static assignment $\boxtimes \leftarrow e$ is the same as the location yielded by evaluating e before; i.e.,

<7.9> $yield(s; x \leftarrow e, x) = yield(s, \mathbf{e}).$

- Suppose, in addition, that x is not pointed to by an expression in *PL, i.e.,*

$loc(s, x)$ is isolated,

and that y is an identifier not an alias of x, i.e.,

$$loc(s, x) \neq loc(s, \mathbf{y}).$$

Then the location yielded by the evaluation of y is unaltered by the assignment; i.e.,

<7.10> $\mathit{yield}(s;\, x \leftarrow e,\, y) = \mathit{yield}(s, \mathbf{y})$.

• Futhermore, the value of x after executing the assignment is the same as the value of e before; i.e.,

<7.11> $\mathit{val}(s;\, x \leftarrow e,\, x) = \mathit{val}(s,\, e)$

Proposition $\Rightarrow$ *Lemma:* The proof of the lemma is by induction on the structure of ♦☜ In other words, we assume inductively that the lemma holds for every proper subexpression t' of ♦☜

Let us suppose that the restrictions of the lemma are satisfied for t; thus

(a) x is not an alias of any other identifier that occurs in ♦▭ i.e.,

$\mathit{loc}(s, \mathrm{x}) \neq \mathit{loc}(s, \mathrm{w})$ for all identifiers w in ♦ such that $\mathrm{w} \neq \mathrm{x}$.

(b) $\mathit{loc}(s, \mathrm{x})$ is isolated; i.e.,

$\mathit{loc}(s, \mathrm{x})$ is not accessible from $\mathit{loc}(s, e')$, for any expression e' in *PL.*

Therefore, $\mathit{loc}(s, \mathrm{x})$ satisfies the static accessibility property <7.2> :

$\mathit{data}(s;\, \mathbf{x} \leftarrow e,\, a) = \mathit{data}(s,\, a)$

for every location α such that $\alpha \neq \mathit{loc}(s, \mathbf{x})$.

Let us suppose also that

$\mathit{val}(s;\, \mathbf{x} \leftarrow \mathbf{e},\, t,\, \psi) \neq \bot_d$.

It follows that the states we are concerned with are also defined; i.e.,

$s \neq \bot_s$ and $s; \mathbf{x} \leftarrow \mathbf{e} \neq \bot_s$

and that $\mathit{loc}(s, \mathbf{x})$ is a storage location; i.e.,

$\mathit{isstore}(s, \mathit{loc}(s, x))$.

For otherwise, by the illegal axiom for assignment <5.9>,

$$s; \mathbf{x} \leftarrow \mathbf{e} = \bot_s.$$

We attempt to derive the desired conclusion, that

$$val(s;\ x \leftarrow e,\ t,\ \psi) = val(s,\ t \lhd (\mathbf{x} \leftarrow e),\ \psi).$$

The proof distinguishes between several cases, depending on the structure of t.

Case: t is an expression in the domain language DL. Then t contains no identifiers of the programming language *PL,* and $t \lhd (\mathbf{x} \leftarrow e)$ is t itself. Our desired conclusion reduces to

$$val(s;\ x \leftarrow e,\ t,\ \psi) = val(s, t, \psi).$$

But this is true, because the value of any expression in *DL* is independent of the state <6.17>.

Case: t is the identifier x itself. Then $t \lhd (\mathbf{x} \leftarrow e)$ is simply e, and our desired conclusion is

$$val(s;\ x \leftarrow e,\ x,\ \psi) = val(s,\ e,\ \psi).$$

Here, x and e are static expressions, which contain no domain language variables; therefore, by < 6.18 >, their values are independent of the substantiation ψ, and our desired conclusion reduces to

$$val(s;\ \mathrm{x} \leftarrow \mathrm{e},\ \mathrm{x}) = val(s,\ \mathrm{e})$$

But this is precisely the fifth part of the static assignment proposition <7.11>.

Case; t is an identifier y distinct from x. Then $t \lhd (\mathbf{x} \leftarrow e)$ is simply **y, and** our desired conclusion is

$$val(s; \mathbf{x} \leftarrow \mathbf{e},\ \mathbf{y},\ \psi) = val(s,\ \mathbf{y},\ \psi).$$

Because y is a static expression, its value is independent of the substantiation ψ, by <6.2>. In other words, our conclusion reduces to

$$val(s;\ x \leftarrow e,\ y) = val(s,\ y),$$

i.e., (by the definition of value <3.10>),

$$data(s;\, x \leftarrow e,\ yield(s;\, x \leftarrow e,\ y)) = data(s, yield(s,\ y)).$$

But

$$data(s;\, x \leftarrow e,\ yield(s;\, x \leftarrow e,\ y)) = data(s;\, x \leftarrow e,\ yield(s,\ y))$$

by the static assignment proposition < 7.7 >

$$= data(s, yield(s,\ y))$$

by the static accessibility property <7.2> .

Note that in this last step we have relied on the fact that $loc(s,\ x)$ is isolated, and therefore not accessible from $yield(s, y)$.

Case: t is of form $f(t_1, t_2, \ldots, t_n)$, *where* f *is a pure construct in the programming language PL, and* $t_1, t_2, \ldots t_n$ *are ezpressions in the assertion language AL.* Then our desired conclusion is

$$\begin{aligned} &val(s;\, x \leftarrow e,\, f(t_1, t_2, \ldots, t_n), \psi) \\ &\quad = val(s,\, f(t_1, t_2, \ldots, t_n) \triangleleft \langle x \leftarrow e\rangle, \psi), \end{aligned}$$

i.e. (by properties of substitution),

$$\begin{aligned} val(s;\, x \leftarrow e,\, f(t_1, t_2, \ldots, t_n), \psi) = val(s, f(&t_1 \triangleleft \langle x \leftarrow e\rangle, \\ &t_2 \triangleleft \langle x \leftarrow e\rangle, \\ &\vdots \\ &t_n \triangleleft \langle x \leftarrow e\rangle), \psi) \end{aligned}$$

But, by the extension of the *val* operator to the assertion language <6.9>, this amounts to showing that

$$\begin{aligned} f_d(&val(s;\, x \leftarrow e, t_1, \psi) \\ &val(s;\, x \leftarrow e,\, t_2, \psi) \\ &\vdots \end{aligned}$$

$$\begin{aligned} val(s;\, \mathbf{x} \leftarrow \mathbf{e},\, t_n,\, \psi)) \\ &= f_d(val(s, t_1 \triangleleft \langle \mathbf{x} \leftarrow \mathbf{e}\rangle, \psi), \\ &\quad val(s, t_2 \triangleleft \langle \mathbf{x} \leftarrow \mathbf{e}\rangle, \psi), \\ &\quad val(s, t_n \triangleleft \langle \mathbf{x} \leftarrow \mathbf{e}\rangle, \psi)). \end{aligned}$$

But this is true, because $t_1, t_2, \ldots, t_n$ are all subexpressions of t; hence, by the induction hypothesis, we have

$$val(s;\, x \leftarrow e,\, t_i,\, \psi) = val(s, t_i \triangleleft \langle \mathbf{x} \leftarrow \mathrm{e}\rangle,\ \psi) \quad \text{for } i = 1, 2, \ldots, n.$$

Note that we are justified in applying the induction hypothesis in this case: That Conditions (a) and (b) are satisfied is straightforward. Furthermore, we have assumed in the statement of the lemma that

$$val(s; \mathbf{x} \leftarrow \mathbf{e},\, t,\, \psi) \neq \perp_d$$

i.e.,

$$val(s;\, \mathbf{x} \leftarrow \mathbf{e},\, \mathbf{f}(t_1, t_2, \ldots, t_n), \psi) \neq \perp_d.$$

By the extension of the *val* operator to the assertion language **<6.9>,** this **means** that

$$\begin{aligned} f_d(&val(s;\, \mathbf{x} \leftarrow \mathbf{e}, t_1, \psi), \\ &val(s;\, \mathbf{x} \leftarrow \mathbf{e}, t_2, \psi), \\ &val(s;\, \mathbf{x} \leftarrow \mathbf{e}, t_n, \psi)) \neq \perp_d . \end{aligned}$$

It follows, because we have assumed in **<4.2>** that f_d is **a** strict function over the domain, that

$$val(s;\, \mathbf{x} \leftarrow \mathbf{e},\, t_i,\, \psi) \neq \perp_d$$

for each $i, i = 1, 2, \ldots, n$. But this is precisely the condition for applying the induction hypothesis of the lemma.

The cases in which t is a logical expression of form $p(t_1, t_2, \ldots, t_n), t_1 \wedge t_2 \wedge \ldots \wedge t_n, t_1 \vee t_2 \vee \ldots \vee t_n, \neg t'$, etc., are similar to the above *case.*

Case: t is ojform $(\forall w)t'$. Then, because t' is a proper subexpression of t, our induction hypothesis tell us that

$$\begin{aligned}&\text{if } val(s; x \leftarrow e, t', \psi') \neq \perp_d\\ &\text{then } val(s; x \leftarrow e, t', \psi') = val(s, t' \lhd \langle x \leftarrow e\rangle, \psi')\end{aligned}$$

for any s, x, e, and ψ' satisfying the conditions of the lemma. (Note that we have renamed the dummy variable ψ to be ψ' to avoid confusion with the ψ in the conclusion of the lemma).

We would like to show that

$$val(s; x \leftarrow e, (\forall w)t', \psi) = val(s, ((\forall w)t') \lhd \langle x \leftarrow e\rangle, \psi),$$

i.e. (by properties of substitution),

$$val(s; x \leftarrow e, (\forall w)t', \psi) = val(s, (\forall w)(t' \lhd \langle x \leftarrow e\rangle), \psi).$$

We consider two possible subcases:

Subcuse: $val(s; x \leftarrow e, (\forall w)t', \psi) = true$. Then, by the extension of the val operator to the assertion language <6.12>,

$$val(s; x \leftarrow e, t', \langle w \leftarrow d\rangle \circ \psi) = true$$

for every data object d. Therefore, surely,

$$val(s; x \leftarrow e, t', \langle w \leftarrow d\rangle \circ \psi) \neq \perp_d$$

for every data object d. Hence, we can apply our induction hypothesis (taking ψ' to be $\langle w \leftarrow d\rangle \circ \psi$) to deduce that

$$val(s, t' \lhd \langle x \leftarrow e\rangle, \langle w \leftarrow d\rangle \circ \psi) = true$$

for every data object d. It follows (by the extension of the val operator to the assertion language <6.12> again) that

$$val(s, (\forall w)(t' \lhd \langle x \leftarrow e\rangle), \psi) = true.$$

Subcuse: $val(s; x \leftarrow e, (\forall w)t', \psi) = false$. Then, by the extension of the val operator to the assertion language <6.13>,

$$val(s; x \leftarrow e, t', \langle w \leftarrow d_0\rangle \circ \psi) = \text{false}$$

for some data object d_0. Therefore, surely for that data object

$$val(s;\ x \leftarrow e,\ t',\ (w \leftarrow d_0) \circ \psi) \neq \perp_d.$$

Hence, we can apply our induction hypothesis (taking ψ' to be $(w \leftarrow d_0) \circ \psi)$ to deduce that

$$val(s, t' \lhd \langle x \leftarrow e\rangle, (w \leftarrow d_0) \circ \psi) = \text{false}$$

It follows, (by the extension of the *val* operator to the assertion language <6.13>, again), that

$$val(s, (\forall w)(t' \lhd \langle x \leftarrow e\rangle), \psi) = false.$$

In both subcases we have shown that

$$val(s;\ x \leftarrow e,\ (\forall w)t', \psi) = val(s, (\forall w)(t' \lhd \langle x \leftarrow e\rangle), \psi);$$

a third conceivable subcase, in which

$$val(s; x \leftarrow e,\ (\forall w)t',\ \psi) = \perp_d,$$

is excluded by the hypothesis of the lemma.

The case in which t is of form $(\exists w)t'$ is similar to the above case and employs the equivalence between $(\exists w)t'$ and $\neg(\forall w)\neg t'$. The case in which t is a set constructor of form $\{w: t'\}$ is also similar, and employs the extension of *val* to set constructors <6.16>, i.e.,

$$val(s,\ \{w\colon t'\},\ \psi) = \text{the set of all data objects in } D \text{ such that } val(s, t', (w \leftarrow d) \circ \psi) = true$$

If the assertion language includes other constructs, then the lemma must also be proved for these constructs, using the corresponding extension of the *val* operator. ∎

This concludes the proof of the lemma. We still must prove the proposition.

Proof of Proposition:

Suppose that s is a state, x is an identifier, and e is a static expression in *PL* such that

$$isstore(s, loc(s, x)) \text{ and}$$
$$s; x \leftarrow e \neq \perp_s.$$

It follows, by the illegal axiom for assignment <5.9>, that

$$isstore(s, loc(s, x))$$

and, by the undefined axiom for assignment <5.10>, that

$$s; e \neq \perp_s$$

and hence, by <3.13>, that

$$yield(s, e) \neq \perp_\ell.$$

We would like to prove the following five properties:

- $loc(s; x \leftarrow e, y) = loc(s, y)$ for all identifiers y.

But,

$loc(s; x \leftarrow e, y) = loc(s; e, y)$
by the frame axiom for assignment <5.2>
$= loc(s, y)$
by the static expression lemma <4.3> and
the properties of indistinguishability < 3.15 >

- $isstore(s; x \leftarrow e, \ell)$ if and only if $isstore(s, \ell)$ for all locations ℓ.

The proof is similar, by the frame axiom for assignment <5.6>, the static-expression lemma < *4.3* > , and the definition of indistinguishability < 3.20 > .

- $yield(s; x \leftarrow e, x) = yield(s, e).$

Note that, because we have supposed that

$$isstore(s, loc(s, x)),$$

it follows, by the first two parts of this proposition, <7.7> and <7.8>, that

$$isstore(s;\ x \leftarrow e,\ loc(s;\ x \leftarrow e,\ x)).$$

But then

$$\begin{aligned} yield(s;\ x \leftarrow e,\ x) &= store(s;\ x \leftarrow e,\ loc(s;\ x \leftarrow e,\ x)) \\ &\quad \text{by the definition of the} \\ &\quad \text{yield operator for identifiers <3.9>,} \\ &= store(s;\ x \leftarrow e,\ loc(s,\ x)) \\ &\quad \text{by the first part of this proposition <7.7>} \\ &= yield(s,\ e) \\ &\quad \text{by the principal assignment axiom <5.1>.} \end{aligned}$$

• Suppose that, in addition, $loc(s, x)$ is isolated, and that y is an identifier such that

$$loc(s,\ x) \neq loc(s,\ y).$$

We would like to show that

$$yield(s;\ x \leftarrow e,\ y) = yield(s,\ y).$$

The proof distinguishes between two cases, depending on whether or not y is bound to *a* storage location. (If y is bound to the undefined location, the conclusion follows from the strictness of the situational operators.)

Case. $isstore(s, loc(s, y))$. It follows that

$$isstore(s;\ x \leftarrow e,\ loc(s;\ x \leftarrow e,\ y)),$$

by the first two parts, <7.7> and <7.8>, of this proposition. By the definition of the yield operator <3.9>, in this case, our desired conclusion reduces to

$$store(s;\ x \leftarrow e,\ loc(s;\ x \leftarrow e,\ y)) = store(s, loc(s,\ y)).$$

But,

$$\begin{aligned} store(s;\ x \leftarrow e,\ loc(s;\ x \leftarrow e,\ y) \\ &= store(s;\ x \leftarrow e,\ loc(s,\ y)) \\ &\quad \text{by the first part <7.7> of this proposition} \\ &= store(s; e, loc(s,\ y)) \\ &\quad \text{by the frame axiom for assignment <5.3>} \\ &= store(s, loc(s,\ y) \\ &\quad \text{by the static expression lemma <4.3>} \\ &\quad \text{and the definition of indistinguishability} \end{aligned}$$

Case. *not* $isstore(s, loc(s, y))$. It follows that

$$not\ isstore(s; x \leftarrow e, loc(s; x \leftarrow e, y))$$

by the first two parts of this proposition, <7.7> and <7.8>. It also follows, by <3.3>, that y is bound to a data location or a structure location in both state s and state $s; x \leftarrow e$. By the definition of the yield operator in this case <3.8>, our desired conclusion reduces to simply

$$loc(s; x \leftarrow e, y) = loc(s, y),$$

which is the first part <7.7> of this proposition.

• Suppose again that $loc(s, x)$ is isolated. We want to show that

$$val(s; x \leftarrow e, x) = val(s, e).$$

By the corollary <4.3> to the static expression lemma, our desired conclusion in this case reduces to

$$data(s; x \leftarrow e, yield(s; x \leftarrow e, x) = data(s, yield(s, e)),$$

because x and e are both static expressions. But

$$\begin{aligned} & data(s; x \leftarrow e,\ yield(s; x \leftarrow e, x)) \\ & \quad = data(s; x \leftarrow e, yield(s, e)) \\ & \qquad \text{by part <7.9> of this pioposition} \\ & \quad = data(s, yield(s, e)) \\ & \qquad \text{by the static-accessibility} \\ & \qquad \text{property <7.2>,} \\ & \qquad \text{because } loc(s, x) \text{ is isolated} \end{aligned}$$

This concludes the proof of the proposition.

We have shown in this section that, if an appropriate set of restrictions is satisfied, the Hoare assignment axiom is true and can be proved as a theorem. In some simple programming languages these restrictions will always be satisfied and we can apply the assignment theorem with no second thoughts. In the more complex programming languages we find in practice, we can still apply the theorem if we can manage to prove that the restrictions are satisfied.

In languages for which the Hoare axiom does hold, the complexity of deductions will be greatly reduced by application of the above theorem. In general, for languages with similarly regular properties, we can shorten proofs by establishing these properties as theorems. The full power of the situational calculus is required only for languages without such regular properties.

In the next section, we will deal with a more powerful class of expressions for which the assignment theorem does not hold.

8. ASSIGNMENT EXPRESSIONS

Now let us consider the class of "assignment expressions" obtained by freely intermixing the assignment operation and the pure constructs. This class is a subclass of the expressions allowed in several programming languages, such as the ALGOL dialects and LISP.

More precisely, we define the ***assignment expressions*** by the following rules:

- Any identifier x in *PL* is an assignment expression.
- If f is a pure construct and $e_1, e_2, \ldots, e_n$ are assignment expressions, then $f(e_1, e_2, \ldots, e_n)$ is an assignment expression.
- If x is an identifier and e is an assignment expression, then $x \leftarrow e$ is an assignment expression.
- If $e_1, e_2, \ldots, e_n$ are assignment expressions, then $e_1; e_2; \ldots; e_n$ is an assignment expression.

For example,

$$x \leftarrow ((y \leftarrow 2);(x \leftarrow (x+1))) + (y \leftarrow (z-1)) + z$$

is an assignment expression.

Thus, in an assignment expression one assignment statement may occur on the right-hand side of another. For this reason, the Hoare axiom fails to hold for general assignment expressions. However, we can prove the following proposition, which allows us to shorten many deductions concerning assignment expressions.

First, let us define a *left* identifier of an expression e to be one that occurs on the left-hand side of an assignment in e , and a right identifier to be one that occurs anywhere else. Thus, in the above example, x and y are left identifiers, and x and z are right identifiers. Note that the same identifier can be both a left and a right identifier.

Proposition (assignment expression): Suppose s is a state, x an identifier, and e and e' are assignment expressions in *PL,* such that x is not pointed to by any expression in *PL, i.e.,*

$$loc(s, x) \text{ is isolated,}$$

and the evaluation of the assignment $\mathbf{x} \leftarrow$ e is defined, i.e.,

$$s; \mathbf{x} \leftarrow \mathbf{e} \neq \perp_s.$$

Then we have

- *principal assignment*

<8.1> $\mathit{val}(s; x \leftarrow e, y) = \mathit{val}(s, e)$

if y is an alias of x , i.e., if $\mathit{loc}(s, x) = \mathit{loc}(s, y)$;

- *value of assignment*

<8.2> $\mathit{val}(s, x \leftarrow e) = \mathit{val}(s, e)$;

- *frame' assignment*

<8.3> $\mathit{val}(s; x \leftarrow e, e') = \mathit{val}(s; \mathbf{e}, e')$

if, for every right identifier $\mathbf{z}$ of e', $\mathit{loc}(s, x) \neq \mathit{loc}(s, \mathbf{z})$

- ***frame expression***

<8.4> $\mathit{val}(s; e, e') = \mathit{val}(s, e')$

if, for every left identifier y in e and every right identifier $\mathbf{z}$ in $\mathbf{e'}$, $\mathit{loc}(s, y) \neq \mathit{loc}(s, \mathbf{z})$ and $\mathit{loc}(s, y)$ is isolated.

We are not going to prove this proposition; the argument is reminiscent of the previous section. However, we will present some counterexamples to indicate why the restrictions on this proposition are required.

Let us suppose that the following configuration exists in state s:

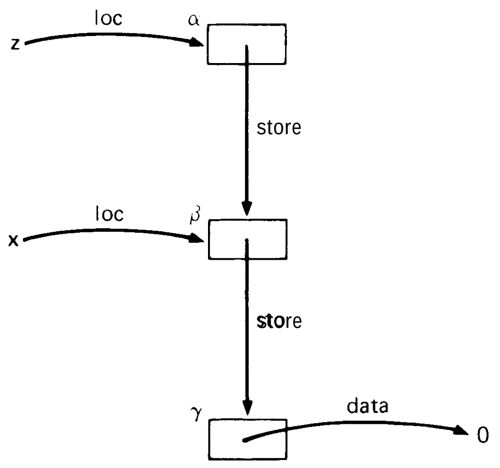


Figure 8.1

Here $loc(s, x)$ is not an isolated location, and the value of **z** is 0, i.e.,

$$val(s, \mathbf{z}) = 0.$$

If we execute the assignment $x \leftarrow \mathbf{z}$ in this state, we create the configuration illustrated below:

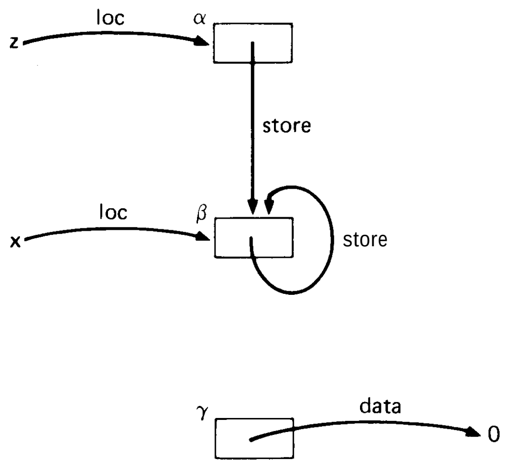


Figure 8.2

Here the identifier x is bound to a looping data structure. We have not yet defined the **data** object represented by a looping data structure, but it is fair to assume that

$$val(s;\ x \leftarrow \mathbf{z},\ x) \neq 0.$$

In other words, we have obtained a contradiction to the conclusion of the principal assignment proposition $val(s;\ x \leftarrow e,\ y) = val(s, e)$ <8.1> by taking e to be **z** and y to be x . Similar contradictions are obtained from this example for the conclusions to the other parts of the assignment expression proposition. In **fact,** any configuration in which an identifier is assigned a structure that points, however indirectly, to that identifier will lead to counterexamples to the proposition.

Another counterexample: the conclusion of the frame expression proposition, $val(s;\ e,\ e') = val(s, e')$,<8.4>, is contradicted immediately if we take e to be $\mathbf{x} \leftarrow x + 1$ and e' to be x; i.e.,

$$val(s;\ x \leftarrow x + 1,\ x) \neq val(s,\ x).$$

Here a condition for the proposition was violated because a left identifier **x** of e is an alias of a right identifier x of e' (in fact, they are identical).

Example:

Let us use the assignment expression proposition to characterize the effect of the assignment expression

$$x \leftarrow ((y \leftarrow (y+2)) + x),$$

in a state s in which the identifiers x and y are bound to isolated storage locations that represent numerical values: We suppose that $val(s, 2) = 2$ and that the programming language construct + is a pure construct associated with the ordinary function + over the numbers. (Thus we ignore the vagaries of computer arithmetic.).

(a) $val(s; x \leftarrow ((y \leftarrow (y + 2)) + x), x) = val(s, x) + val(s, y) + 2$
if $loc(s, x) \neq loc(s, y)$

(b) $val(s; x \leftarrow ((y \leftarrow (y + 2)) + x), y) = val(s, y) + 2$
if $loc(s, x) \neq loc(s, y)$

(c) $val(s; x \leftarrow ((y \leftarrow (y + 2)) + x), x)$
$= val(s; x \leftarrow ((y \leftarrow (y + 2)) + x), y) = 2 \cdot val(s, y) + 4$
if $loc(s, x) = loc(s, y)$.

We will prove both (a) and (c); the proof of (b) is similar.

Proof of (a): We assume $loc(s, x) \neq loc(s, y)$. Then

$val(s; x \leftarrow ((y \leftarrow (y + 2)) + x), x)$
$= val(s, ((y \leftarrow (y + 2)) + x))$
by the principal assignment proposition <8.1>

$= val(s, y \leftarrow (y + 2)) + val(s; y \leftarrow (y + 2), x)$
by the pure-value axiom <4.2>

$= val(s, y + 2) + val(s; y \leftarrow (y + 2), x)$
by the value of assignment proposition <8.2>

$= val(s, y) + val(s, 2) + val(s; y \leftarrow (y + 2), x)$
by the corollary to the static-expression lemma <4.5>

$= val(s, y) + 2 + val(s; y \leftarrow (y + 2), x)$
by our supposition

$= val(s, y) + 2 + val(s, x)$
by the frame expression proposition <8.4>
because $loc(s, y) \neq loc(s, x)$ and $loc(s, y)$ is isolated.

Proof of (c): We assume $loc(s, \mathbf{x}) = loc(s, \mathbf{y})$. Then

$val(s; \mathbf{x} \leftarrow ((y \leftarrow (y + 2)) + x), x)$
$= val(s, y) + 2 + val(s; y \leftarrow (y + 2), x)$
as in the proof of (a)

$= val(s, y) + 2 + val(s, \mathbf{y} + 2)$
by the principal assignment proposition <8.1>,
since $loc(s, x) = loc(s, y)$

$= val(s, \mathbf{y}) + 2 + val(s, \mathbf{y}) + 2$
as in the proof of (a)

$= 2 \cdot val(s, \mathbf{y}) + 4$

These computations are tedious, but purely mechanical. ∎

DISCUSSION

This concludes our treatment of the simple assignment statement; in Parts II and III of this paper we will apply the same approach to data structures and procedure calls. We will defer a full discussion of this approach and a comparison with other approaches until the end of Part III. At this point, however, we can say a few words comparing the situational calculus to both early and contemporary work in the description of programming languages.

The situational calculus was employed by McCarthy [1962] and Burstall [1969] to describe subsets of ALGOL-60. However, the approach has not been further developed or pursued in program verification systems, perhaps because of the apparent relative simplicity of the Floyd/Hoare approach.

The denotational approach of Scott and Strachey (see, e.g., Gordon [1979]) resembles the situational-calculus approach in scope. In the denotational approach, the meaning of a program is represented by a formula of the lambda calculus, whose treatment involves the manipulation of lambda expressions. The situational calculus avoids such expressions, and can therefore exploit such first-order theorem-proving techniques as unification. For this reason, we find the situational calculus more amenable to implementation in automatic verification and synthesis systems.

Acknowledgements

We would like to thank Dan Berry, Mike Gordon, Wolfgang Polak, and Richard Schwartz for discussions valuable in formulating the ideas in this paper; and also Tadashi Kanamori, Yoni Malachi, Bill Scherlis, and Pierre Wolper for their careful reading and perceptive comments. Special thanks are due to Connie Stanley for typesetting the manuscript. This document was prepared on Don Knuth's T_EX text formatting system.

REFERENCES

• **Burstall [1969]**
R.M. Burstall, 'Formal description of program structure and semantics in first order logic," in *Machine* Intelligence *5* (B. Meltzer and D. Michie, eds.), Edinburgh University Press, Edinburgh, 1969, pp. 79-98.

• **Burstall [1972]**
R.M. Burstall, "Some techniques for proving correctness of programs which alter data structures," in Machine *Intelligence 7* (B. Meltzer and D. Michie, eds.), Edinburgh University Press, Edinburgh, 1972, pp. 23-50.

• Cartwright and Oppen [1978]
R. Cartwright and D. Oppen, " Unrestricted procedure calls in **Hoare's** logic," Proceedings of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Ariz. (Jan. 1978), pp. 131-140.

• Clarke [1977]
E.M. Clarke, Jr., "Programming language constructs for which it is impossible to obtain 'good' Hoare-like axiom systems," Proceedings of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, **Calif.** (Jan. 1977), pp. 10-20.

• DeMillo et *al.* [1979]
R.A. DeMillo, R.J. Lipton, and A.J. Perlis, "Social processes and proofs of theorems and programs," CACM, Vol. 22, No. 5 (May 1979), pp. 271-280.

. Floyd [1967]
R.W. Floyd, "Assigning meanings to programs," in the Proceedings of the Symposium on Applied Mathematics, Vol. 19 (J.T. Schwartz, ed.), Providence, RI, 1967, pp. 19-32.

• Gordon et al. [1979]
M. Gordon, R. Milner, C. Wadsworth, *Edinburgh* ***LCF,*** Springer-Verlag, Berlin, 1979.

• Gordon [1979]
M.J.C. Gordon, The *Denotational Description of* ***Programming*** *Lunguugea,* **Springer-**Verlag, Berlin, 1979.

• Green [1969]
C. Green, "Application of theorem proving to problem solving," Proceedings of

the International Joint Conference on Artificial Intelligence, Washington, **D.C.** (May **1969),** pp. 219-239.

• **Hoare** [1969]
C.A.R. Hoare, "An axiomatic basis for computer programming," **CACM, Vol. 12,** No. 10 **(1969),** pp. **576-580.**

• Hoare[1975]
C.A.R. **Hoare,** "Recursive data structures," International Journal of Computer and Information Science, Vol. 4, No. 2 (June 1975).

• Knuth [1974]
D.E. Knuth, "Structured programming with 'go to' statements,* Computing Surveys, **Vol.** *6, No. 4* (Dec. **1974),** pp. **261-301.**

• Kowaltowski [1979]
T. Kowaltowski, "Data structures and correctness of programs," **JACM, Vol.** 26, No. 2 (Apr. **1979),** pp. **283-301.**

• Levy [1978]
M. Levy, "Data types with sharing and circularity," Ph.D. Thesis, University **of** Waterloo, Waterloo, Ontario (May 1978).

• Ligler [1975]
G .T. Ligler, "A mathematical approach to language design", Proceedings of **Second** ACM Symposium on Principles of Programming Languages, Palo Alto, **Calif. (Jan 1975),** pp. 41-53

• London et al. [1978]
R.L. London, J.V. **Guttag,** J.J. Horning, B.W. **Lampson,** J.G. Mitchell, G.J. Popek, "Proof rules for the programming language Euclid,* **Acta Informatica, Vol.** 10, No. 1 **(1978),** pp. l-26.

• Manna and Waldinger [1980]
Z. Manna and **R.** Waldinger, "A deductive approach to program synthesis," **ACM** Transactions on Programming Languages and Systems, Vol. 2, No. 1 (Jan. **1980),** pp. 90-121.

• McCarthy [1962]
J. McCarthy, "Towards a mathematical science of computation," in *Information* Processing, Proceedings of IFIP Congress (CM. Popplewell, ed.), North-Holland, Amsterdam, 1962, pp. 21-28.

- McCarthy [1964]

J. McCarthy, "A formal description of a subset of ALGOL," Report, Stanford University, Stanford, Calif. (Sept. 1964).

- Pratt [1976]

V. R. Pratt, "Semantical considerations on Floyd-Hoare logic," Proceedings of the 17th IEEE Symposium on Foundations of Computer Science (Oct. 1976), pp. 109-121.

- Poupon and Wegbreit [1972]

J. Poupon and B. Wegbreit, "Covering functions," Report, Center for Research in Computing Technology, Harvard University, Cambridge, Mass. (1972).

- Schwartz and Berry [1979]

R. L. Schwartz and D. M. Berry, "A semantic view of ALGOL 68," Computer Languages, Vol. 4 (1979), pp. 1-15.