CMU-CS-84-118

A FULLY ABSTRACT SEMANTICS AND A PROOF SYSTEM FOR AN ALGOL-LIKE LANGUAGE WITH SHARING

Stephen D. Brookes
Computer Science Department
Carnegie-Mellon University
Pittsburgh
Pennsylvania 15213

July 1984

The research reported in this paper was supported in part by funds from the Computer Science Department of Carnegie-Mellon University, and by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539. The views and conclusions contained in it are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

0. Abstract.

In this paper we discuss the semantics of a simple block-structured programming language which allows *sharing* or *aliasing.* Sharing, which arises naturally in procedural languages which permit certain forms of parameter passing, has typically been regarded as problematical for the semantic treatment of a language. Difficulties have been encountered in both denotational and axiomatic treatments of sharing in the literature. Nevertheless, we find that it is possible to define a clean and elegant formal semantics for sharing. The key to our success is the choice of semantic model; we show that conventional approaches based on *locations* are less than satisfactory for the purposes of reasoning about partial correctness, and that in a well defined sense locations are unnecessary in a formal treatment of our programming language. In the first part of the paper we describe a denotational semantics for an ALGOL-like language which allows sharing; the semantic model is not based on locations, but instead uses an abstract *sharing relation* on identifiers to represent the notion of aliasing, and uses an abstract state with a stack-like structure to capture the semantics of blocks. The semantics is shown to be *fully abstract* with respect to partial correctness properties, in contrast to conventional location-based models. This means that the semantics identifies terms if and only if they induce identical partial correctness behaviour in all program contexts. This property typically fails for location-based semantics because in such models it is possible to distinguish between terms on the basis of their effect on individual locations, which has no bearing on partial correctness behaviour.

In the second part of the paper we demonstrate that our choice of semantic model enables us to design a Hoare-style proof system for our language, and to give relatively straightforward proofs of the soundness and completeness of this system. We claim that our proof rules are conceptually simpler to understand than other rules proposed in the literature, without losing any expressive power. Indeed, we are able to *derive* some of these proposed rules. We believe that semantically based methods such as ours may lead to improvements and clarifications in the axiomatic and denotational treatment of other programming constructs. The methods by which we construct a proof system will lead us to some general principles which ought to be more widely applicable. We show, for example, that it is possible to define a "generic" inference rule for blocks which is uniformly applicable to blocks headed by different forms of declaration. The important point here is that, unlike the proof systems for these constructs in the literature, we do not have to design a separate rule for blocks for each possible form of declaration. This results in greater flexibility and adaptability in our proof system. In summary, we are advocating a *generalization* of Hoare logic to encompass the semantics of a wider range of syntactic categories, and we believe that axiomatic semantics should be based directly on, and even derivable from, a suitable choice of semantic model.

1. Introduction.

The need for storage allocation to be explained in programming language semantics arises when we allow blocks to claim "new" storage at run-time. Some authors [11,12] have claimed that conventional denotational semantics based on complete partial orders is incapable of providing an adequate treatment of storage allocation, specifically because of apparent problems in the modelling of storage overflow. The difficulties may appear even more worrying when the storage discipline underlying the programming language allows aliasing or sharing, and when storage de-allocation as well as allocation is intended. All of these issues arise in implementations of ALGOL-like languages, where a block entry causes allocation of some storage which (usually implicitly) becomes de-allocated on block exit. Storage overflow occurs when a program is executed in a store which does not contain enough room for the program's storage claims. A good semantics should be able to model this situation accurately. It has been claimed [11,12] that conventional denotational semantics, based on complete partial orders and continuous functions, cannot achieve these aims, because of an apparent "discontinuity" of storage allocation. Related technical difficulties were noted earlier by Milne and Strachey [15]. These difficulties have led to suggestions that more highly structured and complicated semantic structures be used, such as the so-called "store models" of [11,12]. However, it is too sweeping to conclude that the fault is inherent in the denotational approach to semantics, and that complete partial orders are insufficient. We feel that it is possible to treat all of these supposedly problematic features adequately and elegantly, within the denotational framework, provided we choose an *appropriate* semantic model.

To set the scene, here is a brief description of the current state of conventional denotational semantics as applied to imperative languages. Most of the published attempts at describing a semantics for block-structured languages have involved fairly complicated semantic structures intended to capture the storage book-keeping which seems to be necessary when maintaining the proper sharing relationships. The reader is referred to [9,23] for example. Usually, a semantics for such a language treats as logically separate objects the *environment* and the *store*. The environment is most commonly thought of as a function from identifiers to *locations*; and the store specifies a *contents* function from locations to values, as well as an *area* function indicating the usage status of all locations. Conventionally, the locations represent an abstraction of the notion of addresses in memory, and the store gives the current contents and usage status of these locations. The environment provides the association of identifiers with locations, so that the value of an expression or the effect of a command will depend in general on both the environment and the store. The splitting of "state" into the two components *environment* and *store* has been the standard method of treating aliasing; two identifiers which share are then bound in the environment to the same location.

Most commonly, the semantics of expressions is provided as a function from expressions to environments and stores and then to values; the domain of values typically contains integers and truth values. The semantics of a command, relative to an environment, is modelled as a *store transformation*. Within this type of semantic framework, treatments of many ALGOL-like languages have been given [9,15,17,23]. The explicit separation of state into store and environment does indeed allow a proper treatment to be given of storage allocation and sharing. However, by mentioning locations explicitly in the semantics, these treatments inevitably allow too many semantic distinctions between programs. Using this type of semantics it is possible to distinguish between programs which have identical effects on the *values* of all program identifiers but differ in their effects on the store. For instance, the declaration

$$\mathbf{new}\ x = 0; \mathbf{new}\ y = 1$$

is conventionally described as binding x to the "first" available (*i.e.* unused) location and y to the "next" one, and this means that we are able to distinguish between the effects of this declaration and those of the version in which the two bindings are performed in reverse order:

$$\mathbf{new}\ y = 1; \mathbf{new}\ x = 0.$$

Intuitively, the order of binding should have no effect on any subsequent evaluation, since the two declared identifiers get initialised to the same *values* in each case. When we are merely concerned with the *partial correctness* properties of programs, we need to know only how the execution of a program will affect the *values* of identifiers; in other words, we need to know the *contents* of the locations but not the *identity* of locations. In such a setting, the structure of the store and the attendant details of storage allocation should be kept invisible. In this well defined sense, locations are indeed "unnecessary" for our purposes [7].

At a suitably high level of abstraction, then, we do not want the storage mechanism or even the identities of individual locations to be accessible to the programmer. In other words, a reasonable semantics should be ignorant of the particular choice of storage model on which it might be based. That is, informally, a reasonable semantics should be *more abstract* than a store-based semantics of the standard type. Of course, we would also want to be able to relate our *abstract* semantics to a more *concrete* implementation, perhaps intending to run our programs on a machine with a particular finite memory capacity and with a particular storage allocation algorithm. But any *proof system* for partial correctness should be kept ignorant of the details of storage allocation: the partial correctness of programs is independent of the precise characteristics of the machines on which the programs are to be executed.

In this paper we will construct a semantics possessing these desirable properties. Technically, our semantics will be *fully abstract* with respect to partial correctness behaviour. Full abstraction [20,21] guarantees that two terms of the language are semantically identical if and only if they are interchangeable in every program context. For us, this concept of full abstraction coincides with the equivalence induced by considering partial correctness assertions. This is not the case with the location-based semantics, where semantically distinct terms can nevertheless satisfy precisely the same partial correctness assertions.

In addition, although this paper will not give details, it will be possible to define and prove a *congruence condition* [23] relating our semantics to an implementation at a lower level of abstraction; traditional location-based semantics fall into this category. This shows that any of these other proposed semantic models is consistent with ours. At lower levels of abstraction, of course, such as a location-based semantics, we will also be able to distinguish between programs which require different amounts of new storage during their executions, but which otherwise have identical effects on identifier values.

Ever since Hoare's influential paper [13], which proposed an axiomatic basis for programming languages, many attempts have been made to extend Hoare's methods to more complicated languages. Hoare's paper gave an elegant syntax-directed proof system for an imperative language with (simple) assignment, sequential composition, conditionals, and loops, and introduced the notion of *partial correctness assertion* which has underlined the methods of axiomatic semantics [1]. The appeal and influence of Hoare's work owes much to the syntax-directed nature of his logic and the simplicity of his assertion language.

Many authors have tried to extend Hoare's ideas to cover more complicated and powerful programming constructs [1,2,3,4,14,18]. As we observed earlier, existing proof rules for aliasing seem to be fairly complicated in form [1,2,3,18]. The complications are all the more evident in the case of proof rules for features such as array assignment and procedure calls (see [1,2] for example). We believe that many of the difficulties encountered when trying to find an adequate axiomatization for programming language constructs are caused not by any *inherent* complexity of the construct's semantics but by an inappropriate choice of semantic model or by an inappropriate choice of assertion language. This is particularly true for imperative languages in which storage allocation and the block discipline have persuaded semanticists that the correct level of abstraction should retain some of the details of the storage mechanism. This tends to result in axiom systems in which explicit reasoning about the identity of locations needs to be carried out. And it often happens that some proof rules which appear to be obviously sound are still difficult to prove correct. Apt [1] gives some notable examples. By choosing the appropriate level of abstraction in our semantics we will find not only that it becomes easier to reason about the semantics of programs but that we can find a very simple and obviously sound Hoare-style proof system for the language. The semantics will guide us to a choice of assertion language and proof rules. Locations will not be needed as part of the assertion language, and they will not be necessary in the semantics.

Although we only consider a very simple programming language with a small number of program constructs, we believe that our methods will extend (with suitable modifications in the choice of semantic model) to much more powerful languages including various other features, including loops, conditionals, procedures with various forms of parameter mechanisms, and concurrent or parallel composition. We feel, however, that to include extraneous features at this stage would merely confuse the issues. By focussing on a small number of features and their interactions we aim to clarify the central issues which arise in treatments of sharing, without having to keep extracting the crucial points from a larger setting.

Outline.

The outline of the paper is as follows. We begin by introducing the syntax of our programming language, together with a few relevant syntactic definitions. An informal description of the proposed semantics is given at this stage. Next we identify some general principles behind the semantic treatment of an imperative language, and use them to decide on an appropriate semantic model. We then give a denotational semantics for the language and use it to deduce some useful properties.

Next we define a fairly natural notion of *program behaviour* which captures precisely our intention to concentrate purely on partial correctness properties. Intuitively, two programs should have the same behaviour if they always satisfy the same set of partial correctness assertions. In making these ideas precise, we define the behaviour of a command in a program context. We define a *behavioural equivalence* relation on terms from our language, which identifies two terms if and only if they yield the same behaviour in all program contexts. We then show that our *semantics* induces precisely this relation, so that we have indeed a fully abstract semantics with respect to this notion of behaviour.

We then develop a Hoare-style axiom system for our language, and prove its soundness with respect to our semantics. The proof system is also complete in the usual sense [6]. We give some examples to illustrate the use of the system.

In the final section of the paper we draw some conclusions and make some suggestions for future research.

We provide two Appendices. The first contains definitions and lemmas omitted from the first part of the paper, and sketches proofs of some of the results mentioned there. The second appendix contains a proof of the completeness of the proof system described in the second part of the paper.

2. The Programming Language.

As usual for an imperative language, we distinguish between the following syntactic categories:

$I \in \mathbf{Ide}$	identifiers,
$E \in \mathbf{Exp}$	expressions,
$\Delta \in \mathbf{Dec}$	declarations,
$\Gamma \in \mathbf{Com}$	commands,
$\Pi \in \mathbf{Prog}$	programs.

We assume that the syntax of identifiers and expressions is given; for concreteness, identifiers will be strings of lower-case italic letters. We assume also that it is possible syntactically to determine the identity of two identifiers; we write $I_0 = I_1$ when two identifiers are identical. The syntax of expressions will be ignored; some of our assumptions will become obvious in examples. For instance, an identifier is an expression, and so is a numeral. We assume the usual notions of *free* and *bound* occurrences of identifiers in an expression, and we write free$[\![E]\!]$ for the set of identifiers which occur free in an expression E. An expression having no free identifier occurrences is said to be *closed.*

For the syntax of declarations and commands we specify:

$$\Delta ::= \mathbf{null} \mid \mathbf{new}\, I = E \mid \mathbf{alias}\, I_0 = I_1 \mid \Delta_0; \Delta_1$$
$$\Gamma ::= \mathbf{skip} \mid I{:=}E \mid \Gamma_0; \Gamma_1 \mid \mathbf{begin}\, \Delta; \Gamma\, \mathbf{end}.$$

Informally, we may explain the semantics of these constructs as follows.

Declarations. The purpose of a declaration is to introduce a new set of bindings:

- The **null** declaration has no effect.

- A *simple* declaration of the form

$$\mathbf{new}\, I = E$$

introduces a new binding for I, with initial value the current (declaration time) value of the expression E. We refer to I as the *declared identifier.*

- A *sharing* declaration of the form

$$\mathbf{alias}\, I_0 = I_1$$

introduces a new binding for I_0: the effect of the declaration is to make I_0 *share* with I_1, so that any assignment to I_0 within the scope of this declaration will also affect the value of I_1 (and conversely, an assignment to I_1 within this scope will also update I_0). The declaration also initialises the value of I_0 to the current value of I_1. We refer to I_0 as the declared identifier; note that I_1 is *not* declared here (unless $I_0 = I_1$).

- A *sequential composition* of declarations

$$\Delta_0; \Delta_1$$

accumulates effects from left to right; thus, the scope of Δ_0 in this setting will include Δ_1, but not vice versa. An identifier is declared by $\Delta_0; \Delta_1$ iff either it is declared by Δ_0 or it is declared by Δ_1. If a particular identifier is declared in both Δ_0 and Δ_1 then the latter declaration has precedence.

We adopt the usual notions of free and bound identifier occurrences in declarations, and our informal description above corresponds to the static scope rules familiar from ALGOL-like languages. We refer to

$$\mathrm{dec}[\![\Delta]\!], \quad \mathrm{free}[\![\Delta]\!],$$

as the set of declared (or bound) identifiers and the set of free identifiers of Δ. A declaration without any free identifier occurrences will be called *closed*. For example, the declaration

$$\Delta_0 : \quad \mathbf{new}\, x = 1; \mathbf{new}\, y = x + 1$$

is closed, and clearly its effect is to initialise the new copy of x to 1 and then y to 2. We have $\mathrm{dec}[\![\Delta_0]\!] = \{ x, y \}$ and $\mathrm{free}[\![\Delta_0]\!] = \emptyset$. On the other hand, the declaration

$$\Delta_1 : \quad \mathbf{new}\, y = x + 1; \mathbf{new}\, x = 1$$

contains a free occurrence of x, and the value it gives to y will depend on the current (declaration-time) value of this free identifier. Here we have $\mathrm{dec}[\![\Delta_1]\!] = \{ x, y \}$ and $\mathrm{free}[\![\Delta_1]\!] = \{ x \}$.

Commands. The purpose of a command is to alter the values of identifiers by computing new values for them:

- The **skip** command has no effect.

- An *assignment* $I{:=}E$ updates the value of I to the current (execution-time) value of the expression E. This also has the effect of altering the values of all identifiers which share with I.

- Sequential composition of commands is denoted by $\Gamma_0; \Gamma_1$. The intention is first to perform Γ_0 and then to perform Γ_1, so that again effects accumulate from left to right.

- Finally, a *block*

$$\textbf{begin}\ \Delta; \Gamma\ \textbf{end}$$

allows the *block body* Γ to be executed within the scope of a declaration Δ. This means that the values computed within the block may be affected by the bindings introduced in the declaration; but the scope of the declaration does not extend outside the block, and these bindings are only used locally inside the block. Thus, the semantics of the block as a whole will not involve any changes in *bindings*, only changes in *values* as the result of assignments inside the block.

Examples.

1. The command $x{:=}x+1; y{:=}y+1$ first increases the value of all identifiers which share with x, and then increases the value of all identifiers which share with y; if x and y share, this of course will add 2 to the value of both x and y.

2. The block command

```
begin
   new y = 0;
   x:=x + 1;
   y:=y + 1
end
```

increments the value of identifiers which share with x; the assignment to the local identifier y has no effect outside of the block.

3. The block

```
begin
  alias $z = x$;
  $z:=z + 1$;
  $y:=y + 1$
end
```

has the same effect as the command in Example 1, because the local identifier z shares with the external identifier x. ▮

Again we use the usual syntactic notions of *free* and *bound* occurrences of identifiers in a command. We intend static scoping, as usual in an ALGOL-like language. In particular, for a block we specify

$$\text{free}[\![\textbf{begin } \Delta; \Gamma \textbf{ end}]\!] = \text{free}[\![\Delta]\!] \cup (\text{free}[\![\Gamma]\!] - \text{dec}[\![\Delta]\!]).$$

A complete definition of free$[\![\Gamma]\!]$ for all $\Gamma \in$ **Com** appears in the Appendix.

Programs.

Now we give the syntax for a very simple form of program. A *program* has the following form:

$$\Pi ::= \textbf{begin } \Delta; \Gamma; \textbf{ result } E \textbf{ end},$$

where Δ is a closed declaration containing bindings for all of the free identifiers of Γ and E. In other words, a program contains no free identifier occurrences. This syntactic constraint is reasonable and is commonly imposed in practical programming languages; it will ensure that the effect of a program execution is uniquely determined. The semantics of a program will be represented by the value of the expression E after executing the declarations and command; in other words, we are interested in the *result* of executing a program. Since all of the free identifiers of E are bound in this context, we will see that there is no ambiguity about this value.

3. Semantics.

Preliminaries.

We can define a semantics for our language as follows. The value of an expression may depend on the values of its free identifiers. An association of values with identifiers can be represented as a *valuation*, which is simply a function

$$\sigma : \mathbf{Ide} \to V,$$

where V is the set of expression values; for concreteness, we assume that V contains (at least) the integers and truth values, although the precise structure of this set is not crucial to our development.

The effect of a command will be to alter the values of some identifiers (and hence affect the values of expressions); this amounts to a change in the valuation function. Precisely which identifiers get altered will depend on the sharing relation, which should clearly be an *equivalence relation* ρ on identifiers. The sharing relation can be modified by the execution of a declaration.

A sharing relation can be thought of as a function from identifiers to sets of identifiers:

$$\rho : \mathbf{Ide} \to \mathcal{P}(\mathbf{Ide}),$$

satisfying the usual conditions for an equivalence relation:

$$\begin{aligned} &I \in \rho(I), && \text{(reflexivity)}\\ &I_1 \in \rho(I_0) \;\Rightarrow\; I_0 \in \rho(I_1), && \text{(symmetry)}\\ &I_1 \in \rho(I_0) \;\&\; I_2 \in \rho(I_1) \;\Rightarrow\; I_2 \in \rho(I_0). && \text{(transitivity)} \end{aligned}$$

But it must be the case that identifiers in the same sharing class have the same value. It is therefore more appropriate to think of $\langle \rho, \sigma \rangle$ together, with the following *consistency* condition:

$$I_1 \in \rho(I_0) \;\Rightarrow\; \sigma(I_1) = \sigma(I_0). \qquad \text{(consistency)}$$

We will refer to a combination $\langle \rho, \sigma \rangle$ of this type, satisfying these conditions, as a *frame*. We let R be the set of all equivalence relations on Ide and Σ be the set of all valuations. We use F for the set of all frames, and let f range over F. For convenience, when we write f for a frame, it is implicit that ρ and σ are the components, and similarly f' has components ρ' and σ'; this convention also extends to subscripted terms f_i.

In order to cope properly with the block structure of our programming language, we will need to be able to distinguish the "local" frame used inside a block from the "global" frame in force outside of the block; this will allow us to treat properly a "hole in scope" created for a local identifier whose scope is delimited by the block but for which there is a corresponding global identifier. When we enter a block, a new frame should be created to represent the effect of the declaration which is executed at the head of the block; this new frame is in a sense an extension of the previous frame, and an assignment to an identifier inside the block will in general have an effect on the external frame as well as the local frame. When the block is exited, the original frame structure should be restored, although of course some alterations may have been made in the valuation. As is well known for statically scoped languages such as ours, the block discipline can be modelled or implemented with *stacks.* In our case, we can cope by introducing a special abstract form of stack, built up from frames of the type we have already introduced. This informal discussion might suggest that the structure of a stack be:

$$\langle\langle \rho_n, \sigma_n\rangle, \langle \rho_{n-1}, \sigma_{n-1}\rangle, \ldots, \langle \rho_0, \sigma_0\rangle\rangle, \qquad (n \geq 0).$$

But we also need to maintain the *links* between the successive levels of a stack. We will therefore use as stacks structures of the form

$$\langle\langle \rho_n, \sigma_n\rangle, \tau_n, \langle \rho_{n-1}, \sigma_{n-1}\rangle, \ldots, \tau_1, \langle \rho_0, \sigma_0\rangle\rangle, \qquad (n \geq 0),$$

where each τ_i is a function from identifiers into sets of identifiers. The intention is that for an identifier I, the set $\tau_k(I)$ should be the set of identifiers in the $(k-1)^{\text{th}}$ frame whose values (in that frame) are affected by an assignment to I in the k^{th} frame. The τ functions make explicit the links between identifiers declared at various block levels in a command. In addition to the consistency condition which we imposed on a single frame, we also require the following *link consistency* properties (for each k):

$$\begin{aligned} &\forall I' \in \rho_k(I).(\tau_k(I) = \tau_k(I')), && \text{(link consistency)} \\ &\forall I' \in \tau_k(I).(\sigma_k(I) = \sigma_{k-1}(I')). \end{aligned}$$

In fact, a link will map an identifier either to the empty set, indicating that the identifier does not correspond to any identifiers in the next frame, or to an entire sharing class of the next frame. The intuitions behind these constraints should be clear.

A stack in which all of these conditions is satisfied (so that all of its frames and links are consistent) will be called a *state*, and we will use S to stand for the set of states, with typical member s. We use analogues of the usual stack operations *push, unpush, pop*:

- push$(f, \tau)s$ produces a stack with a new top and a corresponding new top link;

- unpush(s) removes the top frame and its link;

- pop(s) produces the top frame (without its link).

Standard properties of these operations will be assumed; for instance,

$$\text{unpush}(\text{push}(f,\tau)(s)) \;=\; s.$$

We define some useful operations on frames and stacks. First, if we wish to alter the value of an identifier, we must also alter the values of the identifiers in its sharing class. Accordingly, we introduce the notation

$$\sigma + [X \mapsto v]$$

to denote the valuation which agrees with σ at all arguments except those in the set X, which it maps to the value v. Algebraic properties of this operation are fairly straightforward and will be used without proof.

In order to explain the semantics of a declaration, we need an operation which introduces a new sharing class while maintaining the conditions for an equivalence relation. We therefore introduce the notation

$$\rho + [X]$$

to denote the relation ρ' given by:

$$\begin{aligned} \rho'(x) &= X, && \text{if } x \in X, \\ &= \rho(x) - X && \text{otherwise.} \end{aligned}$$

It is easy to verify that if ρ is an equivalence relation then so is $\rho + [X]$.

It is also convenient to combine these two operations, giving a function

$$\text{alter} : \mathcal{P}(\mathbf{Ide}) \times V \to [F \to F]$$

which alters the sharing relation and updates the frame accordingly to maintain consistency. The definition is:

$$\text{alter}(X,v)\langle\rho,\sigma\rangle \;=\; \langle \rho + [X], \sigma + [X \mapsto v]\rangle.$$

It is straightforward to verify that if the frame $\langle\rho,\sigma\rangle$ is consistent then so is $\text{alter}(X,v)\langle\rho,\sigma\rangle$.

We may also extend the updating operations to a stack, in the obvious way, so that the required consistency conditions are maintained and so that updates are propagated through the τ links. If s is the stack

$$s \;=\; \langle\langle\rho_n,\sigma_n\rangle, \tau_n, \ldots, \tau_1, \langle\rho_0,\sigma_0\rangle\rangle,$$

then we define $\text{update}(I,v)s$ to be the stack arising from s by altering the valuations at all relevant levels, beginning at the top by setting the value of (the sharing class of) I to

v, and then updating in a similar way the values at the next level of all identifiers linked to I; this operation applies at all levels of the stack which are accessible through the links from I. Thus,

$$\text{update}(I, v)(s) = \langle\langle \rho_n, \sigma'_n \rangle, \tau_n, \ldots, \tau_1, \langle \rho_0, \sigma'_0 \rangle\rangle,$$

where for each i,

$$\sigma'_i = \sigma + [X_i \mapsto v],$$

with the sets X_i given by

$$\begin{aligned} X_n &= \rho_n(I), \\ X_{i-1} &= \tau_i(X_i) = \bigcup\{\, \tau_i(I') \mid I' \in X_i \,\}. \end{aligned}$$

Note that this stack updating operation does not affect the sharing classes or links of the stack.

Let us write $f(I) = \langle \rho(I), \sigma(I) \rangle$. We say that two frames $f = \langle \rho, \sigma \rangle$ and $f' = \langle \rho', \sigma' \rangle$ *agree* on an identifier I if $f(I) = f'(I)$. This notion extends in the obvious way to agreement on a set X of identifiers. By definition, agreement requires that the two frames agree both on the sharing classes and the values of the selected identifiers. The notion of agreement also extends in the obvious way to a stack. Two stacks s and s' agree on I if their top frames agree on I, their top links agree on I, and if the stacks unpush(s) and unpush(s') agree on all identifiers linked to I.

A sharing class will be called *trivial* if it consists of a single identifier. We will be particularly concerned with frames in which there are only finitely many non-trivial sharing classes, and in which all sharing classes are finite sets. This motivates the following definition.

Definition. A frame f is *finitary* if it contains only finitely many non-trivial sharing classes and all of its sharing classes are finite. A stack s is finitary if all of its frames are finitary. ∎

A Denotational Semantics.

We are now ready to give the semantics. As usual for a denotational presentation, the definitions follow the syntax of the language; there is one semantic clause for each construct, the meaning of a compound term being built up from the meanings of its parts.

Identifiers. The meaning of identifiers is supplied explicitly by a frame, which gives the sharing class and the value of each identifier. In a frame $f = \langle \rho, \sigma \rangle$, the *sharing class* of I is $\rho(I)$; the *value* of I is $\sigma(I)$.

Expressions. As we remarked earlier, the semantics of expressions is taken for granted; it is assumed that expressions do not cause side-effects, so that the only important semantic feature of an expression is its *value*. We assume that all expressions, once supplied with values for their free identifiers, denote elements of a set V. The semantic function will be denoted

$$\mathcal{E} : \mathbf{Exp} \rightarrow [\Sigma \rightarrow V].$$

For any expression E and any valuation σ we refer to $\mathcal{E}[\![E]\!]\sigma$ as the value of E in σ. The only clause of importance at this point is the one for identifiers. We assume that the value of an identifier is supplied by the valuation:

$$\mathcal{E}[\![I]\!]\sigma = \sigma(I).$$

We also assume that the value denoted by an expression depends only on the values of its free identifiers. This is a standard property of statically scoped expression languages. To be precise, our development depends on:

Assumption 1. If two valuations σ and σ' agree on all free identifiers of an expression E, then

$$\mathcal{E}[\![E]\!]\sigma = \mathcal{E}[\![E]\!]\sigma'. \quad \blacksquare$$

Corollary. If an expression E is closed its value is independent of the valuation: for all σ and σ',

$$\mathcal{E}[\![E]\!]\sigma = \mathcal{E}[\![E]\!]\sigma'. \quad \blacksquare$$

The only important property of the set V on which our results will depend is that every element of this set is indeed the value of some expression in Exp. Thus:

Assumption 2. For all $v \in V$, there exists a (closed) expression E_v with value v: for all σ,

$$\mathcal{E}[\![E_v]\!]\sigma = v. \quad \blacksquare$$

It would be straightforward to adapt our arguments to a particular expression language, provided the semantics satisfies the Assumptions above. By isolating the important semantic properties of expressions in this way, without being explicit about the syntax of expressions, we are able to prove some general results which are applicable to a wide variety of expression languages.

Declarations. A declaration produces a new frame, together with a linking function which shows the relationship between identifiers in the new (local) frame and the old (global) frame. We define a semantic function

$$\begin{aligned} \mathcal{D} &: \mathbf{Dec} \to [F \to (F \times T)], \\ \text{where} \quad T &= [\mathbf{Ide} \to \mathcal{P}(\mathbf{Ide})]. \end{aligned}$$

If f is an initial frame, then $\mathcal{D}[\![\Delta]\!]f$ will define a new frame and a link. In fact, it is convenient to factor this semantic function into components by first defining

$$\mathcal{F} : \mathbf{Dec} \to [F \to F]$$

with semantic clauses:

$$\begin{aligned} \mathcal{F}[\![\mathbf{null}]\!]f &= f \\ \mathcal{F}[\![\mathbf{new}\, I = E]\!]f &= \text{alter}(\{ I \}, \mathcal{E}[\![E]\!]\sigma)f \\ \mathcal{F}[\![\mathbf{alias}\, I_0 = I_1]\!]f &= \text{alter}(\rho(I_1) \cup \{ I_0 \}, \mathcal{E}[\![I_1]\!]\sigma)f \\ \mathcal{F}[\![\Delta_0; \Delta_1]\!]f &= \mathcal{F}[\![\Delta_1]\!](\mathcal{F}[\![\Delta_0]\!]f). \end{aligned}$$

This semantic function describes the effect of a sharing relation on frames, showing how the new frame is built up from the old one. The definition of the linking semantics is straightforward. We define the semantic function $\mathcal{T}$, of type

$$\mathcal{T} : \mathbf{Dec} \to [T \to T],$$

as follows:

$$\begin{aligned} \mathcal{T}[\![\mathbf{null}]\!]\tau &= \tau \\ \mathcal{T}[\![\mathbf{new}\, I = E]\!]\tau &= \tau + [I \mapsto \emptyset] \\ \mathcal{T}[\![\mathbf{alias}\, I_0 = I_1]\!]\tau &= \tau + [I_0 \mapsto \tau(I_1)] \\ \mathcal{T}[\![\Delta_0; \Delta_1]\!]\tau &= \mathcal{T}[\![\Delta_1]\!](\mathcal{T}[\![\Delta_0]\!]\tau). \end{aligned}$$

This function builds the link between the old and new frames. The relationship between these semantic functions is simply:

$$\mathcal{D}[\![\Delta]\!]f = \langle \mathcal{F}[\![\Delta]\!]f, \mathcal{T}[\![\Delta]\!]\rho \rangle.$$

Thus, the old sharing relation ρ is to be used in defining the link; it is clear from the definition that an identifier declared by an *alias* form of declaration links with the sharing class of the identifier on the right-hand side of the declaration, whereas a *new* declaration is strictly local in the sense that there is no link between the declared identifier and the old sharing relation. This is achieved by linking the identifier with the empty set.

The following properties of declarations are deducible from the above definitions: that all declarations preserve the consistency and the finitary nature of frames. This shows that our semantic functions are well defined. The proofs are simple structural inductions.

Lemma 1A. For all Δ and all finitary frames f, $\mathcal{F}[\![\Delta]\!]f$ is finitary. ∎

Lemma 1B. For all Δ and all consistent frames f, $\mathcal{F}[\![\Delta]\!]f$ is consistent. ∎

Lemma 1C. For all frames $f = \langle\rho,\sigma\rangle$, $\mathcal{T}[\![\Delta]\!]\rho$ is a consistent link between f and $\mathcal{F}[\![\Delta]\!]f$.

Proof. Show by structural induction that if τ links f and f' then $\mathcal{T}[\![\Delta]\!]\tau$ links $\mathcal{F}[\![\Delta]\!]f$ and f'. Since ρ links $\langle\rho,\sigma\rangle$ with itself trivially, the result then follows. ∎

Note that a declaration has a purely *declarative* semantic aspect, the effect it has on the sharing relation; and an updating or *imperative* aspect, the effect it has on the value of the bound identifiers. Indeed, it will be convenient later to be able to separate these two semantic aspects of declarations. We therefore introduce the functions

$$\mathcal{R} : \mathbf{Dec} \to [R \to R]$$
$$\mathcal{S} : \mathbf{Dec} \to [\Sigma \to \Sigma]$$

defined implicitly by

$$\mathcal{F}[\![\Delta]\!]\langle\rho,\sigma\rangle = \langle\mathcal{R}[\![\Delta]\!]\rho, \mathcal{S}[\![\Delta]\!]\sigma\rangle.$$

It is easy to verify from the definition of $\mathcal{F}$ that both $\mathcal{R}$ and $\mathcal{S}$ are well defined, in the sense that the declarative effect of a declaration depends only on the sharing component of the state, and the imperative effect of a declaration depends only on the valuation. In fact, these two semantic functions could have been defined in the denotational style. The details are simple and appear in Appendix I.

Now that we have a formal definition of the semantics of declarations, we are able to prove some interesting and intuitive properties. For instance, the bindings introduced by a *closed* declaration do not depend on the frame. This is actually a consequence of a more general theorem. If two frames agree on the *free* identifiers of Δ then the bindings introduced by Δ in these frames will be identical:

Theorem 1. For all Δ, if f and f' agree on free$[\![\Delta]\!]$ then for all $I \in \mathrm{dec}[\![\Delta]\!]$,

$$(\mathcal{D}[\![\Delta]\!]f)(I) = (\mathcal{D}[\![\Delta]\!]f')(I).$$

Proof. See Appendix I. ∎

Corollary. The bindings introduced by a closed declaration do not depend on the frame. If Δ is closed, then for all f and f', and all identifiers $I \in \mathrm{dec}[\![\Delta]\!]$,

$$(\mathcal{D}[\![\Delta]\!]f)(I) = (\mathcal{D}[\![\Delta]\!]f')(I). \quad ∎$$

It is also easy to prove that a declaration can *only* alter the value of its *bound* identifiers. This is even true for a sharing declaration. In addition, the link between an identifier which is not redeclared in the new frame and the identifiers in the old frame is simply determined by the sharing relation, as we might expect. The following theorem states these properties precisely.

Theorem 2. For all Δ and all frames $\langle \rho, \sigma \rangle$, if $I \notin \mathrm{dec}[\![\Delta]\!]$ then

$$\begin{aligned}(\mathcal{R}[\![\Delta]\!]\rho)(I) - \mathrm{dec}[\![\Delta]\!] &= \rho(I) - \mathrm{dec}[\![\Delta]\!]\\ (\mathcal{S}[\![\Delta]\!]\sigma)(I) &= \sigma(I),\\ (\mathcal{T}[\![\Delta]\!]\rho)(I) &= \rho(I).\end{aligned}$$

Proof. By structural induction on Δ. ∎

Next we examine the effect of a change in bound identifiers. Let $\langle y\backslash x\rangle\Delta$ denote the result of replacing all *bound* occurrences of x in Δ by y. This syntactic operation can be defined formally, although we relegate the details to Appendix I. For instance,

$$\begin{aligned}\langle y\backslash x\rangle \mathbf{new}\ x = x + z &= \mathbf{new}\ y = x + z\\ \langle y\backslash x\rangle(\mathbf{new}\ x = x + y; \mathbf{alias}\ z = x) &= \mathbf{new}\ y = x + y; \mathbf{alias}\ z = y\\ \langle y\backslash x\rangle(\mathbf{new}\ x = x + y; \mathbf{new}\ y = z) &= \mathbf{new}\ y = x + y; \mathbf{new}\ y = z.\end{aligned}$$

Of course, if x is not declared in Δ then it cannot occur bound in Δ, and $\langle y\backslash x\rangle\Delta$ is syntactically identical to Δ in this case. When x does occur bound in Δ we have

$$\mathrm{dec}[\![\langle y\backslash x\rangle\Delta]\!] = (\mathrm{dec}[\![\Delta]\!] - \{x\}) \cup \{y\}.$$

If, in addition, y does not occur bound or free in Δ, then we expect the resulting declaration $\langle y\backslash x\rangle\Delta$ to have essentially the same effect on y as Δ has on x.

Examples.

To show the necessity of the constraints on y above, that y should not occur free or bound in the declaration, consider the following two declarations.

$$\begin{aligned}\Delta_0 : &\quad \mathbf{new}\ x = 0; \mathbf{alias}\ y = z; \mathbf{alias}\ a = x,\\ \Delta_1 : &\quad \mathbf{new}\ x = z; \mathbf{alias}\ a = y.\end{aligned}$$

1. Note that y is declared in Δ_0. The effect of this declaration is to make a share with the local identifier x. The renamed version is:

$$\langle y \backslash x \rangle \Delta_0 : \quad \mathbf{new}\ y = 0; \mathbf{alias}\ y = z; \mathbf{alias}\ a = y,$$

and now the effect is different because a and z share.

2. In Δ_1 the identifier y occurs free; in the renamed version the free occurrence becomes bound,

$$\langle y \backslash x \rangle \Delta_1 : \quad \mathbf{alias}\ y = z; \mathbf{alias}\ a = y,$$

and now a fails to share with the external y. ∎

In summary, then, we can see that every declaration affects the values of only a finite set of identifiers, those occurring on the left-hand side of a sub-declaration. Moreover, the values used in these bindings depend only on the free identifiers of the declaration (again, a finite set of identifiers). A renaming of bound identifiers has the expected effect. And any frame arising from an initial finitary and consistent frame by some combination of declarations is itself finitary and consistent.

Commands. We are interested only in the effect a command has on the values denoted by identifiers. This effect will depend on the sharing relation. This would suggest that we define the semantics of commands as a function S of type

$$S : \mathbf{Com} \to [R \to [\Sigma \to \Sigma]].$$

However, during the execution of a command there may be block entrances and exits, which have the effect of modifying and restoring the local bindings of identifiers. This effect is modelled conveniently by an abstract stack. Of course, in general during an execution of a command the stack may contain more than one frame, the precise number being determined by the depth of the nested block structure of the command. In order to specify the effect of a command execution, it is therefore cleaner and simpler to specify a semantic function

$$\mathcal{C} : \mathbf{Com} \to [S \to S],$$

which gives the effect of a command on an arbitrary stack. We will be able to verify that commands do not affect the sharing relations of a state, and we will be able to recover a semantic function $\mathcal{S}$ of the above type in a simple way. Given an initial state s, the result $\mathcal{C}[\![\Gamma]\!]s$ will be a state with the same sharing structure as the original one, giving the final values of all identifiers after execution of the command. In the special case when the stack has a single frame $\langle \rho, \sigma \rangle$, we will abuse notation and write $\mathcal{C}[\![\Gamma]\!]\rho\sigma$ for the result. This will be a frame $\langle \rho, \sigma' \rangle$ in which σ' reflects the changes made by assignments in Γ. This will enable us to give an implicit definition of $\mathcal{S}$.

Where convenient we will abuse notation and write $\mathcal{E}[\![E]\!]s$ for the value of E in the (valuation of the) top frame of s. A similar convention may be adopted for $\mathcal{D}$.

The semantic clauses are:

$$\begin{aligned} \mathcal{C}[\![\mathbf{skip}]\!]s &= s \\ \mathcal{C}[\![I{:=}E]\!]s &= \text{update}(I, \mathcal{E}[\![E]\!]s)s \\ \mathcal{C}[\![\Gamma_0;\Gamma_1]\!]s &= \mathcal{C}[\![\Gamma_1]\!](\mathcal{C}[\![\Gamma_0]\!]s) \\ \mathcal{C}[\![\mathbf{begin}\ \Delta;\Gamma\ \mathbf{end}]\!]s &= \text{unpush}(\mathcal{C}[\![\Gamma]\!](\text{push}(\mathcal{D}[\![\Delta]\!]s)s)). \end{aligned}$$

Note that in the clause for a block, we specify that the block body Γ is to be executed in the scope of the declaration Δ by first pushing the new frame and link created by this declaration onto the stack. By unpushing at the end of the block body's execution, we ensure that the original bindings of the identifiers declared in Δ are restored on exiting the block; this is because the scope of the new bindings introduced in Δ does not extend outside the block, and the local meanings of the declared identifiers inside the block are unrelated to their meanings outside the block. This corresponds to the usual notion of static scope.

Examples.

1. $\mathcal{C}[\![\mathbf{begin}\ \Delta;\mathbf{skip}\ \mathbf{end}]\!]s = \text{unpush}(\text{push}(\mathcal{D}[\![\Delta]\!]s)(s)) = s.$

2. $\mathcal{C}[\![x{:=}x+1]\!]s = \text{update}(\rho(x), \sigma(x)+1)(s),$ where $\langle \rho, \sigma \rangle = \text{top}(s)$. ∎

It should be clear from the definitions that executing a command maintains consistency of the state, in the following sense.

Lemma 2A. For all Γ, and all consistent states s, the state $\mathcal{C}[\![\Gamma]\!]s$ is also consistent.

Proof. By structural induction, using Lemmas 1A, 1B and 1C. ∎

The following result establishes our claim that commands do not affect the sharing structure, by which we mean the local sharing relations in each frame of the state and the links between them.

Lemma 2B. For all Γ, and all consistent states s, the states s and $\mathcal{C}[\![\Gamma]\!]s$ have identical sharing structure.

Proof. By structural induction on Γ. ∎

As a consequence of this, we may indeed define a semantic function

$$\mathcal{S} : \mathbf{Com} \rightarrow [R \rightarrow [\Sigma \rightarrow \Sigma]],$$

with the implicit definition being simply:

$$\mathcal{C}[\![\Gamma]\!]\rho\sigma \;=\; \langle \rho, \mathcal{S}[\![\Gamma]\!]\rho\sigma \rangle.$$

Properties of $\mathcal{S}$ may be proved by first establishing a corresponding property for $\mathcal{C}$. Among the most interesting and useful results are the following.

Firstly, a command can affect the values only of identifiers that share with its free identifiers.

Theorem 3. Let f be a frame. Then for all commands Γ, if $I \not\in \rho(\mathrm{free}[\![\Gamma]\!])$ then the valuations σ and $\mathcal{S}[\![\Gamma]\!]\rho\sigma$ agree on I.

Proof. By structural induction on Γ. ∎

Secondly, the semantics of a command depends only on its free identifiers.

Theorem 4. If frames f and f' agree on $\mathrm{free}[\![\Gamma]\!]$ then the valuations $\mathcal{S}[\![\Gamma]\!]\rho\sigma$ and $\mathcal{S}[\![\Gamma]\!]\rho'\sigma'$ agree on $\mathrm{free}[\![\Gamma]\!]$.

Proof. By structural induction, using Theorems 1 and 3. ∎

Next we state a useful and intuitive property of blocks. This shows that the effect of a block on an identifier that is not redeclared at the head of the block can be calculated entirely in the local frame.

Lemma 3. For all Δ, all Γ, and all frames $\langle \rho, \sigma \rangle$, the valuations

$$\mathcal{S}[\![\mathbf{begin}\,\Delta; \Gamma\,\mathbf{end}]\!]\rho\sigma \quad \text{and} \quad \mathcal{S}[\![\Gamma]\!](\mathcal{R}[\![\Delta]\!]\rho)(\mathcal{S}[\![\Delta]\!]\sigma)$$

agree on all identifiers $I \not\in \mathrm{dec}[\![\Delta]\!]$. ∎

If we perform a syntactic substitution on Γ, renaming all free occurrences of x to y, we obtain the command $[y\backslash x]\Gamma$. As usual, this operation incorporates appropriate renamings of bound identifiers to avoid captures. A full definition appears in Appendix I. For example, we have

$$[y\backslash x](x{:=}x+1;\ z{:=}4) \;=\; (y{:=}y+1;\ z{:=}4)$$
$$[y\backslash x](\textbf{begin new } a = x;\ a{:=}x+y \textbf{ end}) \;=\; (\textbf{begin new } a = y;\ a{:=}y+y \textbf{ end}).$$

If y does not occur free in Γ, then $[y\backslash x]\Gamma$ contains an assignment to y precisely where Γ has an assignment to x. Intuitively, then, the effect of executing $[y\backslash x]\Gamma$ should be similar to that of Γ except that (the sharing class of) y is updated instead of (the sharing class of) x. In particular, if the initial state agrees on x and y, so that the two identifiers belong to the same sharing class and have the same initial value, and if $y \not\in \text{free}[\![\Gamma]\!]$, we would also expect the final value of y after executing $[y\backslash x]\Gamma$ to be the same as the final value of x after Γ.

It is also possible to prove that a (suitably constrained) change in bound identifiers has no effect on the semantics of a block. This is an intuitively obvious and desirable property of blocks, and will be useful later when we formulate proof rules for our programming language. The following is a formal statement of this "change of bound variables" property.

Lemma 4. If $x \in \text{dec}[\![\Delta]\!]$, $y \not\in \text{dec}[\![\Delta]\!]$, $y \not\in \text{free}[\![\Delta]\!]$, and $y \not\in \text{free}[\![\Gamma]\!]$, then for all valuations σ consistent with ρ,

$$\mathcal{S}[\![\textbf{begin } \Delta;\Gamma \textbf{ end}]\!]\rho\sigma \;=\; \mathcal{S}[\![\textbf{begin } \langle y\backslash x\rangle\Delta;\ [y\backslash x]\Gamma \textbf{ end}]\!]\rho\sigma. \quad \blacksquare$$

Examples.

To show the necessity of the constraints on y in Lemma 4, consider the following examples.

1. Let Δ_0 be the declaration heading the block

```
begin
   alias x = z;
   new y = 0;
   x:=x + 1
end.
```

This contains a declaration for y, although y does not occur free in either the declaration or command of the block. Clearly, the effect of the block is identical to that of the single assignment $z{:=}z+1$. However, when we change the bound identifier from x to y, we get the block

```
begin
   alias $y = z$;
   new $y = 0$;
   $y:=y + 1$
end,
```

which is semantically identical to **skip**, because the assignment now affects only a local identifier of the block. ▮

2. In this example, y occurs free in the declaration, but not bound in the declaration or free in the command:

```
begin
   new $x = 0$;
   alias $z = y$;
   $z:=z + 1$
end.
```

This block is semantically equivalent to the single assignment $y:=y+1$. Again, the renamed version has no effect. ▮

3. In the block

```
begin null; $x:=y + 1$ end
```

y occurs free in the command. This single assignment will not always have the same effect as the renamed version, which is $y:=y + 1$.

4. In each of the above examples, if we choose instead a fresh identifier w which does not occur free or bound in the block, the semantics is preserved by the change in bound identifier. For instance, the first example becomes

```
begin
   alias $w = z$;
   new $y = 0$;
   $w:=w + 1$
end,
```

which is still semantically equivalent to the assignment $z:=z + 1$. ▮

Programs. For programs, we define the semantic function

$$\mathcal{M} : \mathbf{Prog} \to V,$$

where V is the set of expression values. Recall that a program

$$\mathbf{begin}\ \Delta; \Gamma; \mathbf{result}\ E\ \mathbf{end}$$

consists of a closed declaration, followed by a command and an expression, whose free identifiers are all bound by the head declaration. We want the value of this expression to be the result of executing the program. Since Δ is closed, it binds the free identifiers of Γ and E to values *independent* of the initial state (Theorem 1). The execution of Γ affects only (a subset of) these identifiers, and the values used in updates depend only on the initial values of these identifiers, which are supplied by the declaration. Thus, the state produced by executing the program body specifies values for the free identifiers of E which are again independent of the initial state. We may, therefore, define unambiguously the semantics of a program to be:

$$\mathcal{M}[\![\mathbf{begin}\ \Delta; \Gamma; \mathbf{result}\ E\ \mathbf{end}]\!] \;=\; \mathcal{E}[\![E]\!](\mathcal{C}[\![\Gamma]\!](\mathrm{push}(\mathcal{D}[\![\Delta]\!]s_0)s_0)),$$

where s_0 is defined to be the state with a single frame in which all sharing classes are trivial and in which all identifiers have some dummy value.

For example, the program

$$\mathbf{begin}\ \mathbf{new}\ x = 0; \mathbf{new}\ y = x + 1; y{:=}y + 1; \mathbf{result}\ y\ \mathbf{end}$$

has result 2.

Note that the initial state s_0 is finitary and consistent. Using Lemmas 1A, 1B, 1C, 2A and 2B it is easy to show that all states arising during a program execution are finitary and consistent. By definition, this means that at all times during an execution there are only finitely many non-trivial sharing classes, each of which is itself finite. And the state is always consistent, so that at each level ρ is always an equivalence relation and σ always agrees on the value of each member of a sharing class; the links between the stack frames are always consistent with the valuations.

From now on, we assume that all states are finitary and consistent. We have shown that no semantic details are lost by making this assumption, because the states occurring in any computation are guaranteed to have these properties.

4. Full Abstraction.

Now that we have defined a semantics for our language, we can use it to define the usual *semantic equivalence* relations. Two commands are semantically equivalent iff they denote the same value:

$$\Gamma_0 \equiv \Gamma_1 \quad \Leftrightarrow \quad \mathcal{S}[\![\Gamma_0]\!] = \mathcal{S}[\![\Gamma_1]\!].$$

Thus, two commands are identified by the semantics iff whenever executed from the same initial frame they produce the same final valuation: for all frames f, $\mathcal{S}[\![\Gamma_0]\!]\rho\sigma = \mathcal{S}[\![\Gamma_1]\!]\rho\sigma$.

Similarly, for the other syntactic categories we can define

$$E_0 \equiv E_1 \quad \Leftrightarrow \quad \mathcal{E}[\![E_0]\!] = \mathcal{E}[\![E_1]\!],$$
$$\Delta_0 \equiv \Delta_1 \quad \Leftrightarrow \quad \mathcal{D}[\![\Delta_0]\!] = \mathcal{D}[\![\Delta_1]\!].$$

Two expressions are equivalent iff they always evaluate to the same value: for all valuations σ, $\mathcal{E}[\![E_0]\!]\sigma = \mathcal{E}[\![E_1]\!]\sigma$. And two declarations are equivalent iff they always introduce the same bindings: for all frames f, $\mathcal{D}[\![\Delta_0]\!]f = \mathcal{D}[\![\Delta_1]\!]f$. Semantic equivalence of identifiers is trivial, coinciding with syntactic identity, so we do not bother to introduce a new notation for it.

Finally, for programs we define

$$\Pi_0 \equiv \Pi_1 \quad \Leftrightarrow \quad \mathcal{M}[\![\Pi_0]\!] = \mathcal{M}[\![\Pi_1]\!].$$

Two programs are equivalent iff their results are the same.

Clearly, each of these relations is an equivalence relation. We would like to be sure that our semantics identifies pairs of terms if and only if they are interchangeable, without affecting the semantics, in all program contexts. In other words, we would like semantic equivalence to coincide with *behavioural equivalence*.

There is, for each syntactic category, a set of *program contexts* suitable for filling by members of that category. For instance, the following are program contexts of type *expression:*

$$\textbf{begin new } x = [\,\cdot\,];\ x{:=}\ 1;\ \textbf{result } 42 \textbf{ end},$$
$$\textbf{begin new } x = 0;\ \textbf{new } y = x + 1;\ y{:=}[\,\cdot\,];\ \textbf{result } y \textbf{ end}$$
$$\textbf{begin new } x = 0;\ \textbf{new } y = x + 1;\ y{:=}y + 1;\ \textbf{result } [\,\cdot\,] \textbf{ end}.$$

It is possible, but not particularly illuminating, to define rigorously a syntax for program contexts of these types. We omit the details; see [20] for example. We will use the notation $\Pi[\cdot]$ for a program context, with the type being inferrable from the usage. We also use the notation $\Pi[\tau]$ for the result of filling the hole of a context with a term τ of the appropriate type. It should be understood that we will only consider this substitution to be defined when the result is indeed a syntactically correct program.

Since we have defined our semantics in the denotational style, we know that semantic equivalence implies behavioural equivalence. In other words, for all Δ_i, all E_i, and all Γ_i,

$$\begin{aligned} \Delta_0 \equiv \Delta_1 &\quad\Rightarrow\quad \forall\Pi[\,\cdot\,].(\Pi[\Delta_0] \equiv \Pi[\Delta_1]), \\ E_0 \equiv E_1 &\quad\Rightarrow\quad \forall\Pi[\,\cdot\,].(\Pi[E_0] \equiv \Pi[E_1]), \\ \Gamma_0 \equiv \Gamma_1 &\quad\Rightarrow\quad \forall\Pi[\,\cdot\,].(\Pi[\Gamma_0] \equiv \Pi[\Gamma_1]). \end{aligned}$$

The converse relations, however, are not so obvious. Does behavioural equivalence guarantee semantic equivalence? It is precisely here that problems arise with location-based semantics. If the semantics includes explicit mention of the locations used by a command, then the two commands

$$\textbf{begin new } x = 0; \textbf{ skip end}, \qquad \textbf{skip}$$

will fail to be semantically equivalent, unless the semantics provides explicitly for the releasing of locally claimed storage on exiting a block. Yet they induce the same behaviour in all program contexts, since neither of them alters the value of any identifier. Similarly, a location-based semantics will fail to identify the two declarations

$$\begin{gathered} \textbf{new } x = 0; \textbf{ new } y = 1, \\ \textbf{new } y = 1; \textbf{ new } x = 0, \end{gathered}$$

which obviously have the same behaviour in all contexts; this will even be true in the case of a deallocating semantics.

Our semantics does identify these pairs of terms. In fact, our semantics is *fully abstract*: it identifies terms *if and only if* they produce identical results in all program contexts. Thus, semantic equivalence coincides with behavioural equivalence.

The full abstraction result depends on a simple *expressivity* property of the expression language **Exp** (*Assumption 2*). For convenience, we restate this assumption here.

Assumption 2. For every $v \in V$ there exists a closed expression $E_v \in \mathbf{Exp}$ such that for all frames f, $\mathcal{E}[\![E_v]\!]f = v$. ∎

Provided the expression language **Exp** satisfies this (very reasonable) condition, we can always define a program which, given a finite piece of information about a state, produces a state consistent with this information during a computation. If two terms have a different semantics in some state, then we can build a program context in which the two terms would induce different behaviours. The important property of terms is that they only depend on and affect the values of *finitely many* identifiers. This was established by Theorems 1, 2, 3, and 4.

Lemma 5. For any finitary frame f and any finite set of identifiers A, there is a declaration Δ_f^A such that for all f'

$$\mathcal{F}[\![\Delta_f^A]\!]f' \text{ and } f \text{ agree on } A.$$

Proof. Let $f(A) = \{\, f(I) \mid I \in A \,\}$ be the set of sharing classes and values determined by A and f. We use an induction on the size of this set.

• If $f(A) = \emptyset$ there is nothing to prove, since this can only happen if A is empty, and all pairs of frames agree trivially on the empty set; in this case, we put $\Delta_f^{\emptyset} = \mathbf{null}$.

• For the inductive step, where $f(A)$ is non-empty, we have $f(A) = \{\, f(I)\,\} \cup f(B)$, for some $I \in A$, where $f(B)$ has smaller size. Let $f(I) = \langle X, v\rangle$. Let the distinct elements of X be $I_1, \ldots, I_k$; the set is finite because f is finitary. By our hypothesis on the expression language, there is a (closed) expression E_v having value v. Define the declaration Δ_f^I to be:

$$\mathbf{new}\, I = E_v;\, \mathbf{alias}\, I_1 = I; \ldots;\, \mathbf{alias}\, I_k = I.$$

Clearly, this declaration places the identifiers $I_1, \ldots, I_k$ into a new sharing class initialised to the value v. Thus, for all f', the states f and $\mathcal{D}[\![\Delta_f^I]\!]f'$ will agree on I. By the inductive hypothesis, there is a declaration Δ_f^B which produces agreement on B. We may put

$$\Delta_f^A = \Delta_f^I; \Delta_f^B.$$

Actually, the order does not matter, and $\Delta_f^B; \Delta_f^I$ would have the same effect. ∎

A similar result for commands may be stated and proved in an analogous manner.

Lemma 6. For any finitary frame $f = \langle \rho, \sigma\rangle$, and any finite set A of identifiers, there is a command Γ_f^A such that for all σ' consistent with ρ

$$\mathcal{S}[\![\Gamma_f^A]\!]\rho\sigma' \text{ and } \sigma \text{ agree on } A.$$

These results may be used to prove the full abstraction theorem:

Theorem 5. The semantic functions $\mathcal{E}$, $\mathcal{D}$, and $\mathcal{S}$ are fully abstract.

Proof.

• For $\mathcal{E}$, by assumption.

• For $\mathcal{D}$, we wish to show that for all declarations Δ_0 and Δ_1,

$$\forall \Pi[\,\cdot\,].(\Pi[\Delta_0] \equiv \Pi[\Delta_1]) \quad \Rightarrow \mathcal{D}[\![\Delta_0]\!] = \mathcal{D}[\![\Delta_1]\!].$$

Suppose that $\mathcal{D}[\![\Delta_0]\!] \neq \mathcal{D}[\![\Delta_1]\!]$. We will construct a program context to distinguish between these two declarations. We know by Theorems 1 and 2 that if two declarations Δ_0 and Δ_1 have different semantics then there is a finitary frame f and an identifier I such that

$$(\mathcal{D}[\![\Delta_0]\!]f)(I) \neq (\mathcal{D}[\![\Delta_1]\!]f)(I).$$

Let $\mathcal{D}[\![\Delta_i]\!]f = \langle\langle\rho_i, \sigma_i\rangle, \tau_i\rangle$, for $i = 0, 1$, and let $f_i = \langle\rho_i, \sigma_i\rangle$ stand for the two frames. We know that either the *values* $\sigma_i(I)$ differ, or the (local) *sharing classes* $\rho_i(I)$ differ, or else the *links* $\tau_i(I)$ differ.

There are three cases to consider. Firstly, if the value of I is different in f_0 and f_1, let $A = \text{free}[\![\Delta_0]\!] \cup \text{free}[\![\Delta_1]\!]$. Using Lemma 5, we can find a declaration Δ_f^A as above. Then the program context

$$\textbf{begin } \Delta_f^A; [\,\cdot\,]; \textbf{skip}; \textbf{result } I \textbf{ end}$$

will distinguish between Δ_0 and Δ_1.

Secondly, if the sharing class of I is different in f_0 and f_1, we can choose an identifier I' which shares with I in only one of the frames f_0, f_1. And we can choose an expression E' to have a different value from the value of I in f_0 and f_1. Let

$$A = \text{free}[\![\Delta_0]\!] \cup \text{free}[\![\Delta_1]\!] \cup \text{free}[\![I' := E']\!].$$

By Lemma 5 there is a declaration Δ_f^A as above. The program context

$$\textbf{begin } \Delta_f^A; [\,\cdot\,]; I' := E'; \textbf{result } I \textbf{ end}$$

will distinguish between Δ_0 and Δ_1.

Finally, if the link differs, there must be an identifier (say, I') linked to I in only one of the two cases. With an appropriate choice of A, the context

$$\textbf{begin } \Delta_f^A; \textbf{begin } [\,\cdot\,]; I := I + 1 \textbf{ end}; \textbf{result } I' \textbf{ end}$$

will distinguish between Δ_0 and Δ_1.

- For $\mathcal{C}$ a similar argument can be based on Lemmas 5 and 6. ∎

5. Axiomatic Semantics.

We have defined a denotational semantics for our programming language and proved full abstraction with respect to a notion of program behaviour. It should be clear that this notion is closely related to partial correctness. In other words we have built a fully abstract partial correctness semantics for our language. In this section we show how we can use the structure of the semantics to suggest assertion languages for expressing semantic properties of the terms of our programming language, and then build an axiomatic proof system for the language. The choice of assertion languages and the proof rules are suggested directly by the semantics, and this means that soundness and completeness of the proof system are easy to establish. Moreover, the fact that we have defined separate semantic functions for declarations and commands allows us to separate the axiomatic treatment into two parts: an axiomatization of the purely declarative part of our programming language, and an axiomatization of the imperative part of the language. Since the semantic descriptions were *denotational*, i.e. syntax-directed, we will be able to build syntax-directed (Hoare-style) proof systems.

In our semantics for the programming language, declarations had effects on both the sharing relation (a *declarative* effect) and on the association of identifiers to values (an *imperative* effect), because of the initialisations that take place when a declaration is performed. Commands have an effect only on the values of identifiers, and do not alter the sharing relation. At all times during the execution of a program the sharing classes are all finite, and all but finitely many of them are trivial. Moreover, the constitution of each sharing class is syntactically determined: the set of declarations in whose scope a command is executing determines the sharing classes precisely. There is a reasonably obvious notion of when a declaration Δ specifies that the set X of identifiers is a sharing class. We may formalise this notion precisely. The important point that we are making is that we can choose a language of assertions about sharing and build a Hoare-style proof system for declarations. In fact, we will give a Hoare-style system which is sound and complete. Our choice of assertion language will be dictated by the semantics, which will guide us to an assertion language which is *expressive* in Cook's sense. Once we have axiomatized the declarative semantics, we will then be able to construct a Hoare-style proof system for the imperative effects of commands and declarations.

Declarative Proof System.

The purely *declarative* effect of a declaration is to alter the structure of the sharing classes, in the manner described by the semantic function $\mathcal{R}$. Our approach is to choose a simple language of *assertions* about sharing classes. Specifically, an assertion will be a finite set X of identifiers, or more generally a finite conjunction (written as a list) of a disjoint collection of such sets. The intention is that an assertion

$$X_1, \ldots, X_n$$

lists *all* of the non-trivial sharing classes. Since we know that sharing relations enjoy the finitary property, it is certainly possible to find a finite description of a sharing relation as such a list. There is a simple "propositional" calculus of assertions, which we will largely take for granted. In particular, we use juxtaposition for conjunction and we write

$$X_1, \ldots, X_n \Rightarrow Y_1, \ldots, Y_m$$

when each Y_j is included as a member of the first sequence, so that for some i we have $Y_j = X_i$. We also allow this in the case when Y_j is a singleton disjoint from all of the X_i. The interpretation of such an assertion is clear: the list of X_i contains all of the non-trivial sharing classes; Y_j is "implied" by this list iff either Y_j is non-trivial and appears in the list, or else Y_j is trivial. Thus, the two lists describe precisely the same sharing relation; in this sense, "implication" is trivial for our class of assertions.

We will use X, Y, and Z to stand for finite sets of identifiers (*sharing classes*) and ϕ, ψ, for conjunctions of these (*sharing assertions*). It is convenient to introduce the notation

$$\phi(I) = Y$$

to mean that the sharing class of I, specified by ϕ, is Y. Thus, for instance, if ϕ is $\langle X_1, \ldots, X_n \rangle$ and I belongs to X_i, then $\phi(I) = X_i$; if I is not included in any of the listed classes, then $\phi(I) = \{ I \}$.

We introduce the notation $[Y \backslash I]X$ for the result of replacing I by Y in X:

$$\begin{aligned} [Y \backslash I]X &= (X - \{ I \}) \cup Y \quad &\text{if } I \in X, \\ &= X &\text{otherwise.} \end{aligned}$$

This notation extends in the obvious way to a sharing assertion, $[Y \backslash I]\phi$.

We now design a Hoare-style, syntax-directed proof system for declarations. The assertion

$$\langle \phi \rangle \Delta \langle \psi \rangle$$

is interpreted as saying that if ϕ describes the sharing relation before executing the declaration Δ then ψ will describe the sharing relation afterwards. We use angled brackets instead of conventional set brackets merely to indicate that we are axiomatizing the properties of a different syntactic category from the usual one (commands).

We can build a very simple set of axioms and proof rules, as follows. We give one axiom or rule for each syntactic form of declaration.

• An empty declaration, which we represent by **null**, does not alter any sharing classes:

$$\langle\phi\rangle\mathbf{null}\langle\phi\rangle \tag{A1}$$

• A simple declaration produces a new sharing class containing a single identifier; it removes the newly declared identifier from its old sharing class, and all other sharing classes remain unchanged:

$$\langle\phi\rangle\mathbf{new}\, I = E\langle[\emptyset\backslash I]\phi\rangle \tag{A2}$$

Note that we have

$$\begin{aligned}([\emptyset\backslash I]\phi)(I) &= \{\, I \,\},\\ ([\emptyset\backslash I]\phi)(I') &= \phi(I') - \{\, I \,\}, \quad \text{if } I' \neq I.\end{aligned}$$

Thus, our axiom does indeed correspond to the intuitive explanation given above.

• A sharing declaration has slightly more complicated properties. Specifically, the declared identifier is to be inserted in the sharing class of the identifier on the right-hand side of the declaration, while being removed from its old sharing class. Thus, we specify the axiom:

$$\langle\phi\rangle\mathbf{alias}\, I_0 = I_1\langle[\{\, I_0, I_1 \,\}\backslash I_1][\emptyset\backslash I_0]\phi\rangle \tag{A3}$$

Note that when $X' = [\{\, I_0, I_1 \,\}\backslash I_1][\emptyset\backslash I_0]X$, we have

$$\begin{aligned}I_1 \in X \quad &\Rightarrow \quad X' = X \cup \{\, I_0 \,\},\\ I_1 \not\in X \quad &\Rightarrow \quad X' = X - \{\, I_0 \,\}.\end{aligned}$$

Again this corresponds to the intuitive explanation.

• Finally, consider a sequential composition. Since the second declaration is executed within the scope of the first, the effects should accumulate from left to right. The desired rule to capture this is analogous to the usual rule for sequential composition of commands:

$$\frac{\langle\phi\rangle\Delta_0\langle\phi'\rangle \quad \langle\phi'\rangle\Delta_1\langle\psi\rangle}{\langle\phi\rangle\Delta_0;\, \Delta_1\langle\psi\rangle} \tag{A4}$$

• The following rule allows us to "strengthen" pre-conditions and "weaken" post-conditions:

$$\frac{\phi \Rightarrow \phi' \qquad \langle\phi'\rangle\Delta\langle\psi'\rangle \qquad \psi' \Rightarrow \psi}{\langle\phi\rangle\Delta\langle\psi\rangle} \tag{A5}$$

For an example, let $\langle\,\rangle$ denote the assertion which states that there are no non-trivial sharing classes. Then we have

$$\begin{array}{l}
\langle\,\rangle\textbf{new } x = 0\langle\{x\}\rangle \\
\langle\{x\}\rangle\textbf{alias } y = x\langle\{x,y\}\rangle \\
\langle\{x,y\}\rangle\textbf{alias } z = w\langle\{x,y\},\{z,w\}\rangle \\
\langle\{x\}\rangle\textbf{alias } y = x;\ \textbf{alias } z = y\langle\{x,y,z\}\rangle.
\end{array}$$

It should be clear that these axioms correspond very closely with the semantic function $\mathcal{R}$. Indeed, it is easy to formalise the validity notion for our assertions: let us write

$$\rho \models \phi$$

to denote that the sharing relation ρ satisfies assertion ϕ. Formally, this is defined in a manner corresponding to the informal interpretation given earlier: ϕ lists all of the non-trivial sharing classes, so that

$$\begin{array}{lcl}
\rho \models (X_1,\ldots,X_n) & \Leftrightarrow & \forall i.(I \in X_i \Leftrightarrow \rho(I) = X_i) \\
 & & \&\quad \forall I \notin \bigcup_{i=1}^{n} X_i.\,(\rho(I) = \{I\}).
\end{array}$$

Similarly, we define validity of an assertion $\langle\phi\rangle\Delta\langle\psi\rangle$:

$$\models \langle\phi\rangle\Delta\langle\psi\rangle \quad \Leftrightarrow \quad \forall\rho.(\rho \models \phi \text{ implies } \mathcal{R}[\![\Delta]\!]\rho \models \psi).$$

We claim that the proof system is sound and complete. The proof of soundness is a simple structural induction, and the Appendix contains a sketch of the completeness proof.

Theorem 6. (Soundness) For all Δ and all ϕ, ψ,

$$\vdash \langle\phi\rangle\Delta\langle\psi\rangle \quad \text{implies} \quad \models \langle\phi\rangle\Delta\langle\psi\rangle.$$

Theorem 7. (Completeness) For all Δ and all ϕ, ψ,

$$\models \langle\phi\rangle\Delta\langle\psi\rangle \quad \text{implies} \quad \vdash \langle\phi\rangle\Delta\langle\psi\rangle.$$

Note that none of the assertions gives any information about the *values* denoted by any of the sharing classes. We will see that this will not cause a problem; on the contrary, it is a distinct advantage when we come to formulate proof rules for commands. Essentially, we are separating entirely the purely binding effect of a declaration from the initialisation effect it causes. The latter is more properly regarded as a command-like feature, and we will build it into the proof system for commands.

Imperative Proof System.

For commands, we use assertions of a more conventional style. Pre- and post-conditions are drawn from a simple logical language; examples of conditions are

$$x = 3, \qquad x = y \,\&\, y \neq z.$$

We use P and Q to range over conditions. Each condition represents a predicate on the (valuation part of) state. In conventional Hoare logics for simple sequential languages without sharing, assertions of the form $\{P\}\Gamma\{Q\}$ are used and interpreted as follows: whenever Γ is executed from an initial state satisfying P then the final state will satisfy Q. For languages without sharing this is of course natural, since the effect of a command does not depend on any notion of sharing. However, our semantics for commands involved the sharing relation explicitly. We introduce a natural generalisation of Hoare assertions, incorporating a condition or assumption on the sharing relation. The assertion

$$\phi \vdash \{P\}\Gamma\{Q\}$$

states that whenever the command Γ is executed, with ϕ specifying the sharing classes, from an initial valuation satisfying P, then the final valuation will satisfy Q.

For declarations, we observed that the (local) imperative effect of a declaration was uniquely determined by the valuation, and does not depend on the sharing relation. This suggests that we use assertions of the form

$$\{P\}\Delta\{Q\},$$

with the interpretation that when the declaration Δ is executed from an initial valuation satisfying P, the resulting valuation satisfies Q.

We propose the following axioms and rules of inference for the imperative part of our language. As usual, we give a clause for each command construct. However, in addition, we propose rules of inference for the imperative aspects of declarations. This will enable us to give a simple proof rule for blocks. Our prior axiomatization of declarative semantics will be used.

• A **skip** command has no effect, regardless of the sharing relation:

$$\phi \vdash \{P\}\textbf{skip}\{P\} \tag{B1}$$

• An assignment affects the values of all identifiers in the sharing class of the target identifier, and is thus akin to a simultaneous assignment to a set of distinct identifiers. We use the notation $[E\backslash Y]P$ for the simultaneous syntactic replacement in P of all free occurrences of identifiers in Y by the expression E. This is a generalisation of the single substitution operation $[E\backslash I]P$, and coincides with the latter when Y is a singleton set. The desired axiom is:

$$\frac{\phi(I) = Y}{\phi \vdash \{\,[E\backslash Y]P\,\} I{:=}E\{\,P\,\}} \tag{B2}$$

• The rule for sequential composition is again simple. The two commands are to be executed with the same sharing relation, their effects accumulating from left to right.

$$\frac{\phi \vdash \{\,P\,\}\Gamma_1\{\,Q\,\} \qquad \phi \vdash \{\,Q\,\}\Gamma_2\{\,R\,\}}{\phi \vdash \{\,P\,\}\Gamma_1; \Gamma_2\{\,R\,\}} \tag{B3}$$

• For a block beginning with a simple declaration we have to take into account both the declarative and imperative aspects of the declaration, which may affect the execution of the block body. The following rule takes all of these factors into account. It is sound provided none of the bound identifiers in Δ occurs free in P or R:

$$\frac{\{\,P\,\}\Delta\{\,Q\,\} \qquad \langle\phi\rangle\Delta\langle\psi\rangle \qquad \psi \vdash \{\,Q\,\}\Gamma\{\,R\,\}}{\phi \vdash \{\,P\,\}\mathbf{begin}\ \Delta; \Gamma\ \mathbf{end}\{\,R\,\}} \tag{B4}$$

• A null declaration has no effect:

$$\{\,P\,\}\mathbf{null}\{\,P\,\} \tag{B5}$$

• A simple declaration has an effect similar to that of an assignment, and it updates the value of the declared identifier:

$$\{\,[E\backslash I]P\,\}\mathbf{new}\, I = E\{\,P\,\} \tag{B6}$$

• For a sharing declaration, the effect is similar:

$$\{\,[I'\backslash I]P\,\}\mathbf{alias}\, I = I'\{\,P\,\} \tag{B7}$$

• Sequential composition of declarations behaves simply:

$$\frac{\{\,P\,\}\Delta_0\{\,Q\,\}, \qquad \{\,Q\,\}\Delta_1\{\,R\,\}}{\{\,P\,\}\Delta_0; \Delta_1\{\,R\,\}} \tag{B8}$$

So far we do not have any rule corresponding to "change of bound variable." The block rule above only allows us to use pre- and post-conditions which do not involve the bound identifiers of the block. This is as it should be, since these identifiers are redeclared on entry to the block; the meanings of these identifiers inside the block are unrelated to their meaning outside the block (except through the link). We can suppress the need to reason explicitly about the links by changing bound identifiers to avoid hole-in-scope problems. We need, therefore, to be able to prove an arbitrary partial correctness formula for a block if we can first prove a version in which we have renamed some of the bound identifiers. The following is an adaptation of a standard rule from the literature. Let $(y\backslash x)\Delta$ denote, as before, the result of replacing all *bound* occurrences of x in Δ by y. The rule is:

$$\frac{\phi \vdash \{P\}\textbf{begin}\ (y\backslash x)\Delta;\ [y\backslash x]\Gamma\ \textbf{end}\{Q\}}{\phi \vdash \{P\}\textbf{begin}\ \Delta;\Gamma\ \textbf{end}\{Q\}} \tag{B9}$$

provided y does not occur free in Γ or Δ, and y does not occur bound in Δ. The soundness of this rule, and the need for these syntactic constraints, are indicated by our earlier results (Lemmas 3 and 4).

In addition to the above syntactically motivated rules, the following rule should be self-evident. It allows us to use the consistency property of states to conclude from an assertion about a single identifier I a corresponding assertion about all identifiers in its sharing class. Let us use the notation

$$P_I^X = \bigwedge_{I' \in X} [I'\backslash I]P$$

when X is a finite set of identifiers. For example, we have

$$(x = z + 1)_x^{\{x,y\}} = (x = z + 1 \ \&\ y = z + 1).$$

The rule we propose is simply:

$$\frac{\phi(I) = Y}{\phi \vdash (P \Rightarrow P_I^Y)} \tag{B10}$$

Finally, we include a version of the rule of consequence. Note that it is necessary to include the sharing assertion. The rule for commands is as follows, the one for declarations being similar.

$$\frac{\phi \vdash (P \Rightarrow P') \quad \phi \vdash \{P'\}\Gamma\{Q'\} \quad \phi \vdash (Q' \Rightarrow Q)}{\phi \vdash \{P\}\Gamma\{Q\}} \tag{B11}$$

$$\frac{(P \Rightarrow P') \quad \{P'\}\Delta\{Q'\} \quad (Q' \Rightarrow Q)}{\{P\}\Delta\{Q\}} \tag{B12}$$

Derived rules.

The following specialised rules are derivable. These are special cases of our general rules in which we have chosen a specific form for the declaration at the head of a block. They are related to rules in the literature, especially those of [3].

A rule for a block beginning with a *new* declaration can be obtained from the block rules and the rules for *new*:

$$\frac{\langle\phi\rangle\mathbf{new}\ z = E\langle\psi\rangle \qquad \psi(z) = Z \qquad \psi \vdash \{\,[E\backslash Z]P\,\}[z\backslash x]\Gamma\{\,Q\,\}}{\phi \vdash \{\,P\,\}\mathbf{begin\ new}\ x = E;\Gamma\ \mathbf{end}\{\,Q\,\}}, \tag{D1}$$

where z is a fresh identifier chosen not to be free in any of the relevant terms. Specifically, we require that z should not occur free in P or Q or Γ. In fact, even this version of the rule can be simplified further, using (A2), to get:

$$\frac{[\emptyset\backslash z]\phi \vdash \{\,[E\backslash z]P\,\}[z\backslash x]\Gamma\{\,Q\,\}}{\phi \vdash \{\,P\,\}\mathbf{begin\ new}\ x = E;\Gamma\ \mathbf{end}\{\,Q\,\}}, \tag{D2}$$

again provided z does not occur free in P, Q, or Γ.

A similar rule for a block headed by a sharing declaration is also obtainable:

$$\frac{\langle\phi\rangle\mathbf{alias}\ z = y\langle\psi\rangle \qquad \psi(z) = Z \qquad \psi \vdash \{\,[y\backslash Z]P\,\}[z\backslash x]\Gamma\{\,Q\,\}}{\phi \vdash \{\,P\,\}\mathbf{begin\ alias}\ x = y;\Gamma\ \mathbf{end}\{\,Q\,\}}, \tag{D3}$$

where again z is a fresh identifier chosen not to be free in P, Q, or Γ. An important point to notice is the similarity in structure between these derived rules (D1) and (D3) for the two types of declaration.

Notice that if we only allowed purely declarative declarations which do not perform any initialisation, the rules can be simplified because we no longer need any assertions of the form $\{\,P\,\}\Delta\{\,Q\,\}$. In any case, our rules and axioms are arguably cleaner than the alternatives proposed in the literature. Of course, in the absence of (non-trivial) sharing, these rules collapse down to standard rules, as we would expect. In particular, the assignment rule collapses to Hoare's original axiom [13]:

$$\{\,[E\backslash I]P\,\}I{:=}E\{\,P\,\}.$$

Examples.

Example 1. Consider the following command, which we will denote Γ:

$$\begin{array}{l}\textbf{begin}\\ \quad \textbf{new}\, t = x;\\ \quad x{:=}y;\\ \quad y{:=}t\\ \textbf{end}.\end{array}$$

We claim that this command exchanges the value of x and y, regardless of the sharing relation. We can prove an instance of this very easily. Let ϕ be a sharing assertion with $\phi(x) = X$ and $\phi(y) = Y$. We will prove the assertion

$$\langle\phi\rangle \vdash \{\, x = 0 \ \&\ y = 1 \,\}\Gamma\{\, x = 1 \ \&\ y = 0 \,\}.$$

The proof is simple. Firstly, we have

$$\phi \vdash \{\, x = 0 \ \&\ y = 1 \,\}\textbf{new}\, t = x\{\, x = 0 \ \&\ y = 1 \ \&\ t = 0 \,\}.$$

This follows because we have

$$\langle\phi\rangle \vdash \textbf{new}\, t = x\langle\psi\rangle,$$

where $\psi(x) = X - t, \psi(y) = Y - t, \psi(t) = \{\, t \,\}$. This shows that t does not share with x or y. Then we have

$$\psi \vdash \{\, x = 0 \ \&\ y = 1 \ \&\ t = 0 \,\}x{:=}y\{\, x = 1 \ \&\ y = 1 \ \&\ t = 0 \,\}.$$

Finally, we get

$$\psi \vdash \{\, x = 1 \ \&\ y = 1 \ \&\ t = 0 \,\}y{:=}t\{\, x = 1 \ \&\ y = 0 \ \&\ t = 0 \,\}.$$

The result follows by the block rule and the rule of consequence.

Example 2. To illustrate reasoning about sharing, consider the command

$$\textbf{begin alias}\; z = x;\ \textbf{alias}\; y = z;\ y{:=}x + 1\ \textbf{end}.$$

This should have the effect of increasing the value of all identifiers which share with x. We prove this as follows. Let ϕ be a sharing assertion and let $X = \phi(x)$. Let w be an identifier which does not belong to X, so that w does not share with x. The rules for declarations give

$$\vdash \langle\phi\rangle\textbf{alias}\; z = x;\ \textbf{alias}\; y = z\langle\psi\rangle,$$

where $\psi(x) = X \cup \{\, y, z \,\}$. From this, the assignment rule and the consistency rule gives

$$\langle\psi\rangle \vdash \{\, x = w \,\}z{:=}x + 1\{\, x = y = z = w + 1 \,\},$$

since $w \not\in X$. From this, using the rule of consequence, and the block rule, we get

$$\langle\phi\rangle \vdash \{\, x = w \,\}\textbf{begin alias}\; z = x;\ \textbf{alias}\; y = x;\ y{:=}z + 1\ \textbf{end}\{\, x = w + 1 \,\}.$$

That completes the proof.

Soundness and Completeness.

We claim that our imperative proof system is sound and complete. We have already established this for the purely declarative proof system, which is used in building up the imperative system. Now we have to tackle the proof rules for imperative semantics of declarations and commands.

To be precise, we say that an assertion

$$\phi \vdash \{P\}\Gamma\{Q\}$$

is satisfied in a frame f, written

$$f \models (\phi \vdash \{P\}\Gamma\{Q\}),$$

if when the initial valuation σ of f satisfies P and the sharing relation ρ of f satisfies ϕ, then the final valuation $S[\![\Gamma]\!]\rho\sigma$ satisfies Q:

$$\rho \models \phi \ \& \ \sigma \models P \quad \text{implies} \quad S[\![\Gamma]\!]\rho\sigma \models Q.$$

Similarly, a valuation σ satisfies an assertion $\{P\}\Delta\{Q\}$ if the analogous relationship holds:

$$\sigma \models P \quad \text{implies} \quad S[\![\Delta]\!]\sigma \models Q.$$

An assertion is *valid*, denoted

$$\models (\phi \vdash \{P\}\Gamma\{Q\})$$

if it is satisfied in all frames. For declarations we require that the assertion be satisfied in all valuations.

We need to show that all valid assertions are provable, and every provable assertion is valid. As usual, following Cook [6], we are assuming that we can use any true (valid) assertion of the form $\phi \vdash P$ or P as an assumption in a proof. Let Th be the set of valid conditions of this form:

$$(\phi \vdash P) \in \text{Th} \quad \Leftrightarrow \quad \models (\phi \vdash P),$$
$$P \in \text{Th} \quad \Leftrightarrow \quad \models P.$$

We want to prove that all valid assertions are provable *relative to* Th.

Soundness.

The proofs of soundness are straightforward, relying on the Theorems stated earlier about our semantic functions. We prove that each axiom is valid, and that each inference rule preserves validity. It follows that every proof preserves validity, and that every provable assertion is valid.

Theorem 8. For all Δ and all P and Q,

$$\mathbf{Th} \vdash \{P\}\Delta\{Q\} \quad \text{implies} \quad \models \{P\}\Delta\{Q\} \quad \blacksquare$$

Theorem 9. For all Γ and all ϕ, P and Q,

$$\mathbf{Th} \vdash (\phi \vdash \{P\}\Gamma\{Q\}) \quad \text{implies} \quad \models (\phi \vdash \{P\}\Gamma\{Q\}) \quad \blacksquare$$

For example, the soundness of the block rules follow from Lemmas 3 and 4.

Completeness.

We already know that the declarative system is complete. For the imperative system, we can show that "weakest pre-conditions" can be expressed for each syntactic construct in our assertion language. In other words, our assertion language is *expressive.* Essentially, we define weakest pre-conditions with respect to a sharing relation. The Appendix contains proof sketches.

Theorem 10. For all Δ and all P and Q,

$$\models \{P\}\Delta\{Q\} \quad \text{implies} \quad \mathbf{Th} \vdash \{P\}\Delta\{Q\} \quad \blacksquare$$

Theorem 11. For all Γ and for all ϕ, P and Q,

$$\models (\phi \vdash \{P\}\Gamma\{Q\}) \quad \text{implies} \quad \mathbf{Th} \vdash (\phi \vdash \{P\}\Gamma\{Q\}) \quad \blacksquare$$

6. Conclusions.

We constructed a Hoare-style proof system for a simple block-structured language with sharing. The underlying semantic model with respect to which we proved soundness and completeness was location-free, and this fact enabled the proofs to go through with a minimum of complication. Our semantics was novel in that it involved an abstract model of stacks, which provided a clean and elegant way to cope with the scope rules and block structure of our language. Although it may be argued that stacks have an operational flavour, we would counter by pointing out that the same is true of location-based models. It is at least arguable that the use of stacks is a fairly natural way of explaining the effects of block entry and exit, and that in our semantics we have made no operational commitment with regard to "implementation": although the semantics uses abstract stacks to describe the effect of programs, this will not force us to have to implement the language on a stack machine. Despite our decision to abstract away from the notion of location, it should be possible to relate our semantics to a more concrete location-based model for the language, although we do not discuss this in detail here.

An important suggestion arising from our results and technique is that Hoare-style proof systems should be designed not only for imperative languages—as was the case in Hoare's original paper—but that it is advantageous to extend Hoare's principle to syntactic categories other than commands. We designed a Hoare-style proof system for declarations, and found that this helped immensely in the construction of a proof system for commands. Our methods suggest, we feel, a general basis for constructing Hoare-like semantics for even more complicated languages involving sharing, such as languages including array declarations and array assignments. We feel sure that our methods are applicable (with some modifications, of course, in the choice of semantic model and assertion language) to procedural languages involving various parameter mechanisms and even allowing the use of recursion; some steps in this direction have also been taken by Reynolds [22], whose *specification logic* conforms in spirit with our ideas and techniques; an approach to the semantics of block-structured languages based on category theory has been developed by Óles [19], and there appear to be connections between their abstract store models and ours. Olderog [18] also gives a semantic treatment of aliasing based on sharing classes, but we feel that our treatment is somewhat cleaner, and gains in simplicity and clarity by explicitly focussing on the need to axiomatize separately the semantics of declarations and commands. Other related work includes the proof system of [10], based on "store-models"; in contrast to the approach used there, our underlying semantic model is arguably cleaner, and we believe that our proof rules are more natural. Our proof system was also based on first order assertion languages, a property that is more in the spirit of Hoare logics, as outlined in [5].

In principle, it should be possible to build axiom systems for each syntactic category of a programming language, and combine them to get a Hoare-style proof system for the whole language, as we did here for our simple language. An advantage of this approach is that the hierarchical structure of a proof system built in this way will reflect the syntactic structure of the programming language: in the example language considered here, for instance, the declarative system is a subsystem used inside the imperative system, and this corresponds to the fact that declarations can appear as syntactic components of commands. Of course, for more complicated languages, we may need different choices of assertion language; and the axioms and rules may not be as clean as the ones we were able to use here. However, we are confident that the adoption of a more widely based notion of Hoare system will lead to significant improvements in the axiomatic treatment of many programming language constructs. This will be the subject of a series of further investigations.

The work reported in this paper is an attempt to design a clean and mathematically tractable semantics for a programming language, and to use the semantics directly in the design of an assertion language and proof system for reasoning about partial correctness of programs. We believe that many existing programming language features still lack elegant and tractable formal description, and that their axiomatization has been attempted prematurely. We intend to tackle more complicated languages than the one considered in this paper, as we expect our ideas and techniques to be more generally applicable.

Acknowledgements. The author is grateful to Joe Halpern, Albert Meyer and Boris Trakhtenbrot for helpful comments, and to Allen Stoughton for pointing out an error in an earlier definition of $\mathcal{C}$.

7. Appendix I.

This section lists some definitions whose details were omitted from the text of the paper, and contains some of the proofs and lemmas for some of the theorems of the first part of the paper.

Denotational definitions.

The declaration semantic functions have the following denotational descriptions.

$$\begin{aligned}
\mathcal{R} &: \mathbf{Dec} \to [R \to R] \\
\mathcal{R}[\![\mathbf{null}]\!]\rho &= \rho \\
\mathcal{R}[\![\mathbf{new}\, I = E]\!]\rho &= \rho + [\{I\}] \\
\mathcal{R}[\![\mathbf{alias}\, I_0 = I_1]\!]\rho &= \rho + [\rho(I_1) \cup \{I_0\}] \\
\mathcal{R}[\![\Delta_0; \Delta_1]\!]\rho &= \mathcal{R}[\![\Delta_1]\!](\mathcal{R}[\![\Delta_0]\!]\rho)
\end{aligned}$$

$$\begin{aligned}
\mathcal{S} &: \mathbf{Dec} \to [\Sigma \to \Sigma] \\
\mathcal{S}[\![\mathbf{null}]\!]\sigma &= \sigma \\
\mathcal{S}[\![\mathbf{new}\, I = E]\!]\sigma &= \sigma + [I \mapsto \mathcal{E}[\![E]\!]\sigma] \\
\mathcal{S}[\![\mathbf{alias}\, I_0 = I_1]\!]\sigma &= \sigma + [I_0 \mapsto \sigma(I_1)] \\
\mathcal{S}[\![\Delta_0; \Delta_1]\!]\sigma &= \mathcal{S}[\![\Delta_1]\!](\mathcal{S}[\![\Delta_0]\!]\sigma)
\end{aligned}$$

$$\begin{aligned}
\mathcal{T} &: \mathbf{Dec} \to [T \to T] \\
\mathcal{T}[\![\mathbf{null}]\!]\tau &= \tau \\
\mathcal{T}[\![\mathbf{new}\, I = E]\!]\tau &= \tau + [I \mapsto \emptyset] \\
\mathcal{T}[\![\mathbf{alias}\, I_0 = I_1]\!]\tau &= \tau + [I_0 \mapsto \tau(I_1)] \\
\mathcal{T}[\![\Delta_0; \Delta_1]\!]\tau &= \mathcal{T}[\![\Delta_1]\!](\mathcal{T}[\![\Delta_0]\!]\tau).
\end{aligned}$$

Syntactic definitions.

The definitions of *dec* and *free* for declarations are:

$$\begin{aligned}
\mathrm{dec} &: \mathbf{Dec} \to \mathcal{P}(\mathbf{Ide}) \\
\mathrm{dec}[\![\mathbf{null}]\!] &= \emptyset \\
\mathrm{dec}[\![\mathbf{new}\, I = E]\!] &= \{I\} \\
\mathrm{dec}[\![\mathbf{alias}\, I_0 = I_1]\!] &= \{I_0\} \\
\mathrm{dec}[\![\Delta_0; \Delta_1]\!] &= \mathrm{dec}[\![\Delta_0]\!] \cup \mathrm{dec}[\![\Delta_1]\!]
\end{aligned}$$

$$\begin{aligned}
\text{free} &: \mathbf{Dec} \to \mathcal{P}(\mathbf{Ide}) \\
\text{free}[\![\mathbf{null}]\!] &= \emptyset \\
\text{free}[\![\mathbf{new}\, I = E]\!] &= \text{free}[\![E]\!] \\
\text{free}[\![\mathbf{alias}\, I_0 = I_1]\!] &= \{ I_1 \} \\
\text{free}[\![\Delta_0; \Delta_1]\!] &= \text{free}[\![\Delta_0]\!] \cup (\text{free}[\![\Delta_1]\!] - \text{dec}[\![\Delta_0]\!]).
\end{aligned}$$

Semantic Properties.

With these definitions in mind, it is now possible to prove Theorem 1. Actually, it is easier to establish the following more powerful result, from which Theorem 1 follows as a special case.

Theorem I.1. For all Δ, if A is a set of identifiers such that $\text{free}[\![\Delta]\!] \subseteq A$, then for all frames f, f', if f and f' agree on A then $\mathcal{D}[\![\Delta]\!]f$ and $\mathcal{D}[\![\Delta]\!]f'$ agree on A and on $\text{dec}[\![\Delta]\!]$.

Proof. By structural induction on Δ.

- When Δ is null the result is trivial.

- When Δ is $\mathbf{new}\, I = E$, we have:

$$\begin{aligned}
\mathcal{R}[\![\Delta]\!]\rho &= \rho + [\{ I \}], \\
\mathcal{R}[\![\Delta]\!]\rho' &= \rho' + [\{ I \}].
\end{aligned}$$

Since ρ and ρ' agree on A, it follows that $\rho + [\{ I \}]$ and $\rho' + [\{ I \}]$ agree on A and on I, as required. The argument is similar for $\mathcal{S}$ and $\mathcal{T}$, except that *Assumption 1* is used in the argument for $\mathcal{S}$; this is where the inclusion of $\text{free}[\![\Delta]\!]$ in A is used.

- For an *alias* declaration the proof is similar.

- When Δ is a sequential composition $\Delta_0; \Delta_1$ we have:

$$\begin{aligned}
\mathcal{F}[\![\Delta]\!]f &= \mathcal{F}[\![\Delta_1]\!](\mathcal{F}[\![\Delta_0]\!]f), \\
\mathcal{T}[\![\Delta]\!]\rho &= \mathcal{T}[\![\Delta_1]\!](\mathcal{T}[\![\Delta_0]\!]\rho).
\end{aligned}$$

Since $\text{free}[\![\Delta]\!] = \text{free}[\![\Delta_0]\!] \cup \text{free}[\![\Delta_1]\!]$, we have $\text{free}[\![\Delta_0]\!] \subseteq A$. Thus, by the inductive hypothesis, $\mathcal{D}[\![\Delta_0]\!]f$ and $\mathcal{D}[\![\Delta_0]\!]f'$ agree on A and on $\text{dec}[\![\Delta_1]\!]$. Then, replacing A by $A \cup \text{dec}[\![\Delta_0]\!]$, we can use the inductive hypothesis to deduce the desired result. ∎

The corresponding result for commands (Theorem 4) is established in a similar manner. The relevant definition of *free* for commands is:

$$\begin{aligned} \text{free} &: \mathbf{Com} \to \mathcal{P}(\mathbf{Ide}) \\ \text{free}[\![\mathbf{skip}]\!] &= \emptyset \\ \text{free}[\![I{:=}E]\!] &= \text{free}[\![E]\!] \cup \{I\} \\ \text{free}[\![\Gamma_0;\Gamma_1]\!] &= \text{free}[\![\Gamma_0]\!] \cup \text{free}[\![\Gamma_1]\!] \\ \text{free}[\![\mathbf{begin}\ \Delta;\Gamma\ \mathbf{end}]\!] &= \text{free}[\![\Delta]\!] \cup (\text{free}[\![\Gamma]\!] - \text{dec}[\![\Delta]\!]). \end{aligned}$$

It should be clear from the definitions of $\mathcal{R}$ and $\mathcal{T}$ that a sharing class $(\mathcal{R}[\![\Delta]\!]\rho)(I)$ arises from the corresponding link class $(\mathcal{T}[\![\Delta]\!]\rho)(I)$ by removing and deleting identifiers which are *declared* in Δ. This is the reason for the following lemma, useful in proving part of Theorem 2 and Lemma 3.

Lemma I.2. For all Δ and all $I' \notin \text{dec}[\![\Delta]\!]$, and for all I,

$$I' \in (\mathcal{R}[\![\Delta]\!]\rho)(I) \quad \Leftrightarrow \quad I' \in (\mathcal{T}[\![\Delta]\!]\rho)(I).$$

Properties of Substitutions.

We assume given the syntactic operation $[y\backslash x]$ on expressions. The substitution $[y\backslash x]\Delta$, intended to replace all free occurrences of x by y, is defined:

$$\begin{aligned} [y\backslash x]\mathbf{null} &= \mathbf{null} \\ [y\backslash x](\mathbf{new}\, I = E) &= (\mathbf{new}\, I = [y\backslash x]E) \\ [y\backslash x](\mathbf{alias}\, I_0 = I_1) &= (\mathbf{alias}\, I_0 = I_1) && \text{if } I_1 \neq x \\ &= (\mathbf{alias}\, I_0 = y) && \text{if } I_1 = x \\ [y\backslash x](\Delta_0;\Delta_1) &= [y\backslash x]\Delta_0; [y\backslash x]\Delta_1. \end{aligned}$$

The syntactic substitution $\langle y\backslash x\rangle\Delta$, which replaces all bound occurrences of x by y, is defined:

$$\begin{aligned} \langle y\backslash x\rangle\mathbf{null} &= \mathbf{null} \\ \langle y\backslash x\rangle(\mathbf{new}\, I = E) &= (\mathbf{new}\, I = E) && \text{if } I \neq x \\ &= (\mathbf{new}\, y = E) && \text{if } I = x \\ \langle y\backslash x\rangle(\mathbf{alias}\, I_0 = I_1) &= (\mathbf{alias}\, I_0 = I_1) && \text{if } I_0 \neq x \\ &= (\mathbf{alias}\, y = I_1) && \text{if } I = x \\ \langle y\backslash x\rangle(\Delta_0;\Delta_1) &= (\langle y\backslash x\rangle\Delta_0); (\langle y\backslash x\rangle[y\backslash x]\Delta_1). \end{aligned}$$

Using these definitions it is possible to prove the following results. First, note that (as we would expect), the following properties are provable. In each case a simple structural induction suffices.

Lemma I.3. For all Δ and all $x, y \in \mathbf{Ide}$,

$$\begin{aligned}
\mathrm{free}[\![\langle y\backslash x\rangle\Delta]\!] &= \mathrm{free}[\![\Delta]\!] \\
\mathrm{dec}[\![[y\backslash x]\Delta]\!] &= \mathrm{dec}[\![\Delta]\!] \\
\mathrm{dec}[\![\langle y\backslash x\rangle\Delta]\!] &= (\mathrm{dec}[\![\Delta]\!] - \{\, x \,\}) \cup \{\, y \,\}, && \text{if } x \in \mathrm{dec}[\![\Delta]\!] \\
&= \mathrm{dec}[\![\Delta]\!] && \text{otherwise} \\
\mathrm{free}[\![[y\backslash x]\Delta]\!] &= (\mathrm{free}[\![\Delta]\!] - \{\, x \,\}) \cup \{\, y \,\} && \text{if } x \in \mathrm{free}[\![\Delta]\!] \\
&= \mathrm{free}[\![\Delta]\!] && \text{otherwise.}
\end{aligned}$$

The following theorem states a semantic property of this substitution operation. Corresponding results for $\mathcal{R}$ and $\mathcal{S}$ may be formulated in a similar manner.

Theorem I.4. If $x \in \mathrm{dec}[\![\Delta]\!]$, $y \not\in \mathrm{dec}[\![\Delta]\!]$ and $y \not\in \mathrm{free}[\![\Delta]\!]$ then

$$\mathcal{T}[\![\langle y\backslash x\rangle\Delta]\!]\rho = (\mathcal{T}[\![\Delta]\!]\rho) + [y \mapsto (\mathcal{T}[\![\Delta]\!]\rho)(x)] + [x \mapsto \rho(x)].$$

The definition of $[y\backslash x]\Gamma$ is straightforward, except for the renaming of bound identifiers in a block in order to avoid capture of y.

$$\begin{aligned}
[y\backslash x]\mathbf{skip} &= \mathbf{skip} \\
[y\backslash x](I{:=}E) &= I{:=}[y\backslash x]E && \text{if } I \neq x \\
&= y{:=}[y\backslash x]E && \text{otherwise} \\
[y\backslash x](\Gamma_0; \Gamma_1) &= [y\backslash x]\Gamma_0;\ [y\backslash x]\Gamma_1 \\
[y\backslash x]\mathbf{begin}\ \Delta; \Gamma\ \mathbf{end} &= \mathbf{begin}\ [y\backslash x]\Delta;\ [y\backslash x]\Gamma\ \mathbf{end}, && \text{if } y \not\in \mathrm{dec}[\![\Delta]\!] \\
&= [y\backslash x]\mathbf{begin}\ \langle z\backslash y\rangle\Delta;\ [z\backslash y]\Gamma\ \mathbf{end} && \text{if } y \in \mathrm{dec}[\![\Delta]\!], \\
&\qquad \text{where } z \not\in \mathrm{dec}[\![\Delta]\!], \mathrm{free}[\![\Delta]\!], \mathrm{free}[\![\Gamma]\!].
\end{aligned}$$

8. Appendix II: A Completeness Proof.

This section contains a short sketch of a proof of completeness for the proof system of the paper.

We define first a syntactic "strongest post-condition" for declarations and assertions about the sharing relation. If Δ is a declaration and ϕ is an assertion, we define

$$\mathrm{sp}[\![\Delta]\!](\phi)$$

by induction on the structure of the declaration:

$$\begin{aligned}
\mathrm{sp}[\![\mathbf{new}\, I = E]\!](\phi) &= [\emptyset\backslash I]\phi \\
\mathrm{sp}[\![\mathbf{alias}\, I_0 = I_1]\!](\phi) &= [\{ I_0, I_1 \}\backslash I_1]([\emptyset\backslash I_0]\phi) \\
\mathrm{sp}[\![\Delta_0;\Delta_1]\!](\phi) &= \mathrm{sp}[\![\Delta_1]\!](\mathrm{sp}[\![\Delta_0]\!](\phi))
\end{aligned}$$

Recall the notation $[Y\backslash I]\phi$ for the result of replacing I by Y in ϕ. Formally, this is defined by induction on the structure of ϕ. When ϕ is a single set X of identifiers, we simply put

$$\begin{aligned}
[Y\backslash I]X &= (X - \{ I \}) \cup Y \qquad \text{if } I \in X \\
&= X \qquad\qquad\quad \text{otherwise}
\end{aligned}$$

For a conjunction, we put

$$[Y\backslash I](X_1, \ldots, X_n) = ([Y\backslash I]X_1, \ldots, [Y\backslash I]X_n).$$

The intention is that $\mathrm{sp}[\![\Delta]\!](\phi)$ is a strongest post-condition in the usual sense, so that for all ρ, ϕ and Δ,

$$\rho \models \phi \quad \text{iff} \quad \mathcal{R}[\![\Delta]\!]\rho \models \mathrm{sp}[\![\Delta]\!](\phi).$$

The full property is expressed by the following lemma.

Lemma II.1. For all Δ, and all ϕ and ψ,

$$\models \langle\phi\rangle\Delta\langle\psi\rangle \quad \text{iff} \quad \models (\mathrm{sp}[\![\Delta]\!](\phi) \Rightarrow \psi).$$

Proof. By structural induction on Δ. ∎

The usefulness of these strongest post-conditions is summarized as follows. The proof is as usual by structural induction.

Lemma II.2. For all Δ, and all ϕ,

$$\vdash \langle\phi\rangle\Delta\langle\mathrm{sp}[\![\Delta]\!](\phi)\rangle. \quad ∎$$

Next, we define weakest pre-conditions for the imperative constructs. The following syntactic construction will suffice for commands, assuming we can define $\mathrm{wp}[\![\Delta]\!](Q)$.

$$\begin{aligned}
\mathrm{wp}_\phi[\![\mathbf{skip}]\!](Q) &= Q\\
\mathrm{wp}_\phi[\![I{:=}E]\!](Q) &= [E\backslash\phi(I)]Q\\
\mathrm{wp}_\phi[\![\Gamma_1;\Gamma_2]\!](Q) &= \mathrm{wp}_\phi[\![\Gamma_1]\!](\mathrm{wp}_\phi[\![\Gamma_2]\!](Q))\\
\mathrm{wp}_\phi[\![\mathbf{begin}\ \Delta;\Gamma\ \mathbf{end}]\!](Q) &= \mathrm{wp}[\![\Delta]\!](\mathrm{wp}_{\mathrm{sp}[\![\Delta]\!](\phi)}[\![\Gamma]\!](Q)),
\end{aligned}$$

provided $\mathrm{dec}[\![\Delta]\!]$ does not contain any free identifiers of Q. In a case where this constraint is violated, we may simply rename bound variables, obtaining a more generally applicable but slightly less attractive definition

$$\mathrm{wp}_\phi[\![\mathbf{begin}\ \Delta;\Gamma\ \mathbf{end}]\!](Q) = \mathrm{wp}[\![\Delta']\!](\mathrm{wp}_\psi[\![\Gamma']\!](Q)),$$

where $\Delta';\Gamma'$ is a renaming of the block body to avoid binding the free identifiers of Q, and ψ is $\mathrm{sp}[\![\Delta']\!](\phi)$.

For declarations we define

$$\begin{aligned}
\mathrm{wp}[\![\mathbf{null}]\!](Q) &= Q\\
\mathrm{wp}[\![\mathbf{new}\ I = E]\!](Q) &= [E\backslash I]Q\\
\mathrm{wp}[\![\mathbf{alias}\ I_0 = I_1]\!](Q) &= [I_1\backslash I_0]Q\\
\mathrm{wp}[\![\Delta_0;\Delta_1]\!](Q) &= \mathrm{wp}[\![\Delta_0]\!](\mathrm{wp}[\![\Delta_1]\!](Q)).
\end{aligned}$$

The following lemmas show that this syntactic characterisation does indeed define weakest pre-conditions. Again they are proved by structural induction.

Lemma II.3. For all Γ, and all ϕ, P and Q,

$$\models (\phi \vdash \{P\}\Gamma\{Q\}) \quad \text{iff} \quad \models (\phi \vdash (P \Rightarrow \mathrm{wp}_\phi[\![\Gamma]\!](Q))).$$

Lemma II.4. For all Δ, and all P and Q,

$$\models \{P\}\Delta\{Q\} \quad \text{iff} \quad \models (P \Rightarrow \mathrm{wp}[\![\Delta]\!](Q)).$$

Next, we show that weakest pre-conditions can be used in proofs to establish completeness.

Lemma II.5. For all Γ, and all ϕ, P and Q,

$$\mathbf{Th} \vdash (\phi \vdash \{\mathrm{wp}_\phi[\![\Gamma]\!](Q)\}\Gamma\{Q\}.$$

Lemma II.6. For all Δ, and all P and Q,

$$\mathbf{Th} \vdash \{ \mathrm{wp}[\![\Delta]\!](Q) \}\Delta\{ Q \}.$$

The completeness theorems follow immediately. We state the version for commands; the corresponding result for declarations (Theorem 10) is similar, and may be proved in exactly the same way.

Theorem 11. For all Γ, and all ϕ, P and Q,

$$\models (\phi \vdash \{ P \}\Gamma\{ Q \}) \quad \text{implies} \quad \mathbf{Th} \vdash (\phi \vdash \{ P \}\Gamma\{ Q \}).$$

Proof. Suppose that the assertion $(\phi \vdash \{ P \}\Gamma\{ Q \})$ is valid. Then we know that

$$\models (\phi \vdash (P \Rightarrow \mathrm{wp}_{\phi}[\![\Gamma]\!](Q))).$$

This assertion belongs to the set **Th**, so that trivially we have

$$\mathbf{Th} \vdash (\phi \vdash (P \Rightarrow \mathrm{wp}_{\phi}[\![\Gamma]\!](Q))) \tag{1}$$

We also have

$$\mathbf{Th} \vdash (\phi \vdash \{ \mathrm{wp}_{\phi}[\![\Gamma]\!](Q) \}\Gamma\{ Q \}) \tag{2}$$

by Lemma II.5. By the rule of consequence (B11), from (1) and (2) we get

$$\mathbf{Th} \vdash (\phi \vdash \{ P \}\Gamma\{ Q \}),$$

where **Th** is the set of valid conditions. That completes the proof.

9. References.

(LNCS refers to the Springer Verlag Lecture Notes in Computer Science series.)

[1] Apt, K. R., Ten Years of Hoare's Logic: A survey–Part 1, ACM TOPLAS, Vol. 3 pp 431-483 (1981).

[2] de Bakker, J. W., *Mathematical Theory of Program Correctness*, Prentice-Hall 1980.

[3] Cartwright, R., and Oppen, D., The Logic of Aliasing, Acta Informatica 15, pp 365-384 (1981).

[4] Clarke, E. M., Programming language constructs for which it is impossible to obtain good Hoare axioms, JACM 26 pp 129-147 (1979).

[5] Clarke, E. M., The Characterization Problem for Hoare Logics, Proc. Royal Society Meeting (1984); also CMU Technical Report CS-84-109.

[6] Cook, S., Soundness and completeness of an axiom system for program verification, SIAM J. Comput. 7, pp 70-90 (1978).

[7] Donahue, James, Locations Considered Unnecessary, Acta Informatica 8, pp 221–242 (1977).

[8] German, S. M., Clarke, E. M., and Halpern, J. Y., Reasoning about Procedures as Parameters, Proc. Workshop on Logics of Programs, Springer Verlag LNCS 164 (1984).

[9] Gordon, M., *The Denotational Description of Programming Languages*, Springer Verlag 1978.

[10] Halpern, J., A Good Hoare Axiom System for an Algol-like language, Proc. POPL 1984.

[11] Halpern, J., Meyer, A., and Trakhtenbrot, B., The Semantics of Local Storage, or What Makes the Free-list Free?, Proc. POPL 1984.

[12] Halpern, J., Meyer, A., and Trakhtenbrot, B., From Denotational to Operational and Axiomatic Semantics for ALGOL-like languages: An Overview, Proc. 1983 Workshop on Logics of Programs, Springer Verlag LNCS 164 (1984).

[13] Hoare, C. A. R., An Axiomatic Basis for Computer Programming, CACM 12, pp 576-580 (1969).

[14] Hoare, C. A. R., Procedures and parameters: An axiomatic approach, in: *Symposium on Semantics of Algorithmic Languages*, Springer Verlag LNCS 188 (1971).

[15] Milne, R., and Strachey, C., *A Theory of Programming Language Semantics*, Chapman and Hall 1976.

[16] Milner, R., Fully Abstract Models of Typed λ-calculi, Theoretical Computer Science (1977).

[17] Mosses, P. D., The Mathematical Semantics of ALGOL 60, Technical Monograph PRG-12, Oxford University Computing Laboratory, Programming Research Group (1974).

[18] Olderog, E-R., Correctness of Programs with Pascal-like Procedures without Global Variables, 1981 (to appear in Theoretical Computer Science).

[19] Oles, F. J., A Category-theoretic Approach to the Semantics of ALGOL-like Languages, Ph. D. thesis, Syracuse University (August 1982).

[20] Plotkin, G. D., LCF Considered as a Programming Language, Theoretical Computer Science 5, pp 223-255 (1977).

[21] Plotkin, G. D., and Hennessy, M. C. B., Full Abstraction for a Simple Parallel Programming Language, Proc. MFCS 1979, Springer LNCS 74, pp 108-120 (1979).

[22] Reynolds, J., *The Craft of Programming*, Prentice-Hall 1981.

[23] Stoy, J. E., *Denotational Semantics*, MIT Press 1977.