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

New Revised Version
August 1989

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

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.

We believe that axiomatic reasoning about program behaviour should be based directly on a semantic model specifically tailored for that purpose. Moreover, the structure of the semantic model should be used directly to suggest the structure of an assertion language for expressing program properties. It is desirable, therefore, to adopt a semantics with as clean and simple a structure as possible, so that one can use assertions with simple syntactic structure and build a clean and simple axiomatic proof system for program properties. By basing the proof system closely on the underlying semantic model, one is able to use the semantic model directly in establishing the soundness and relative completeness of the proof system; these tasks are made less difficult if the semantics has a simple structure. We illustrate these ideas by applying them to a small programming language, a simple block-structured imperative language which allows sharing or aliasing among identifiers. Although the language is rather simple and is certainly far from being a fully fledged programming language, it exhibits enough semantic features to merit a detailed semantic investigation and serves well to illustrate our methodology for designing an axiomatization. We first define a standard semantics and discuss the full abstraction problem for this language. We define a semantic relation called sharing equivalence, and show that the standard semantics is fully abstract "up to sharing equivalence". We then construct a semantics based directly on sharing equivalence classes, which is fully abstract. We establish some important semantic properties, and use them in designing an axiomatic proof system for partial correctness properties of programs.

1. Introduction.

The first part of the paper introduces the syntax and a standard denotational semantics for a simple block-structured programming language which allows *sharing* or *aliasing*. Sharing arises naturally in procedural languages which permit certain forms of parameter passing (like call-by-reference): two identifiers *share* or are *aliases* if assignment to one affects the value of the other. Here we do not include procedures in our language, but instead include a form of declaration that introduces aliasing among program identifiers explicitly. This allows us to focus on the semantic treatment of aliasing in a simpler context than a fully fledged procedural language. This has the advantage that a satisfactory treatment of aliasing is possible with a relatively straightforward semantics and axiomatic system. Nevertheless, we must admit that in a language with procedures our treatment would need to be modified extensively. Even in this restricted setting we believe that our approach is of interest.

We begin with a conventional semantic model suitable for the proper modelling of aliasing, involving locations, stores, and environments. Identifiers which are aliases are mapped by the environment to the same location, and an assignment to an identifier affects the value stored in the corresponding location. Hence the common description: identifiers which are aliases *share* a memory location. We define the semantics relative to an abstract storage allocation primitive (as usual called *new*), assumed only to possess a natural "newness" property. We discuss several variant semantics, differing only in details of storage handling.

We next state some well known and fairly obvious properties of commands, expressions, and declarations. Their semantics is determined completely by their dependence on and effect upon identifiers occurring (free) in their text. In proving these results we introduce a natural notion of *sharing equivalence* on environment-store pairs, which plays a crucial role in the rest of the paper. In order later to justify the soundness of an inference rule involving change of bound variables in blocks, We introduce *syntactic substitutions* as renamings of identifiers in program terms. With the appropriate notions of substitution of free identifiers

(in commands, declarations, and expressions) and substitution of declared identifiers (in a declaration), the meaning of a term behaves properly with respect to substitution: in particular, the meaning of a block is unchanged if we systematically rename identifiers used in it. We formulate precisely what we mean by proper behavior under substitution, and prove these results using the semantic definitions. These results will be useful later in axiomatizing the programming language.

We are primarily interested in semantics as the basis for axiomatization of partial correctness. It would therefore be desirable if our semantics identifies pairs of program terms if and only if they can be used interchangeably without affecting the partial correctness of a program: i.e. we would like full abstraction with respect to partial correctness behaviour. It is intuitively clear that when reasoning about partial correctness behaviour it is unnecessary to keep track of all details of storage management; the precise locations bound to identifiers are irrelevant, as long as we know which identifiers are aliases and what the (stored) values of identifiers are. It is also obvious that our standard semantics carries around explicit information about locations. It is only in certain details of storage allocation that our standard semantics misses the full abstraction property. Sharing equivalence is the key to achieving full abstraction: two declarations can be substituted for each other in an arbitrary context if and only if they have the same effect up to sharing equivalence. The same is true of commands. We state and prove this rigorously, as well as showing that the meanings of commands and expressions also are uniquely determined by analogous properties. In this analysis we introduce *sharing relations* and *valuations*: a sharing relation is simply an equivalence relation on a finite set of identifiers, and a valuation is a map from these identifiers to values that respects the sharing relation. Two environment-store pairs are sharing equivalent if and only if they determine the same sharing relation and the same valuation. Because the standard meaning of a term is uniquely determined by its dependence on and effect on sharing relations and valuations we can in fact lift the abstraction function to the level of semantic definitions and define an "abstract" semantics manipulating sharing relations and valuations directly. This new semantics is fully abstract (even for declarations) with respect to partial correctness behaviour. It is even possible to give a denotational definition of the abstract semantics. In doing so, we formulate a syntactic notion of aliasing and show that it corresponds precisely to the intended semantic interpretation of aliasing. We note that several variants of the standard location-based semantics, each of which defines a different semantic equivalence relation on terms, are "equivalent" in that they all correspond to the same abstract semantics. This again argues for our decision to base axiomatic treatment on the abstract model.

This abstract semantics is well suited for the purposes of axiomatizing the programming language. In the second part of the paper we use the semantics to support axiomatic reasoning about program properties. The structure of the semantic model suggests what type of structure is required of assertions about program fragments. We build a Hoare-style proof system for partial correctness properties, and we prove soundness and relative completeness of this system. The proof system is built up in a hierarchical manner which reflects the syntactic and semantic structure of the programming language. We first design a proof system for declarations, and then use it in building proof rules for commands. We claim that our proof rules are conceptually simpler to understand than other rules proposed in the literature for aliasing, without losing any expressive power. 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 most of 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. We demonstrate that some well known rules from the literature for blocks can be derived in our system.

1. Introduction.

To set the scene, here is a brief summary of conventional denotational semantics as applied to imperative languages. Most of the published semantics for block-structured languages have involved fairly complicated semantic structures intended to model storage book-keeping. The reader is referred to [10,22,34,37] for example. Following [19,35], a semantics for such a language treats as logically separate objects the *environment* and the *store*. Roughly speaking, declarations modify the environment and commands modify 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, 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 value of an expression or the effect of a command will depend in general on both the environment and the store. In particular, the value of an identifier is obtained as the contents (via the store) of the location bound (via the environment) to the identifier. The intermediary rôle of locations provides the standard method of treating aliasing: two identifiers which share are bound in the environment to the same location (and, consequently, always have the same value).

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 (at least) integers and truth values. The semantics of a command, relative to an environment, is modelled as a *store transformation*. Explicit separation of state into store and environment, with locations playing an intermediary rôle, does indeed allow a proper treatment to be given of storage allocation and sharing. However, by mentioning locations explicitly in the semantics, these treatments may allow too many semantic distinctions between program fragments. Using this type of semantics it is sometimes possible to distinguish between terms which have identical effects on the *values* of all identifiers but differ in their effects on the store, because of storage allocation.

For instance, the declaration

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

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 permuted declaration in which the two bindings are performed in reverse order:

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

Intuitively, the order of binding should have no effect on any subsequent evaluation, since the two declared identifiers get initialized to the same *values* in each case, and in each case there is the same effect on the sharing properties of identifiers: neither x nor y is an alias for any other identifier.

Another issue is raised by the declarations

$$\textbf{new } x = 0;\ \textbf{new } x = 1 \qquad \text{and} \qquad \textbf{new } x = 1.$$

Obviously both have the effect of introducing a single new variable, initialized to 1. However, in the first case two locations are allocated, one of which becomes inaccessible. Unless care

is taken in the semantic definitions, these two declarations will have different meanings because of this. But of course (assuming no limitation on the size of the store) they can be used interchangeably in any program context.

Similarly, 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.

When we are merely concerned with *correctness* properties of programs (either partial or total correctness), we need to know only how the execution of a program will affect the *values* of identifiers. By choosing a semantic model at a slightly higher level of abstraction it becomes unnecessary to worry about the order in which variables are declared if that order is irrelevant. Donahue [8] showed that locations were unnecessary in a semantic treatment of a language which did not permit aliasing, and we argue further that this is even true when aliasing is allowed.

In the first part of the paper we introduce a block-structured programming language and, starting from a standard location-based semantics, we build a semantics in which these ideas are demonstrated. The main idea is to make explicit use of the notion of a *sharing relation* on identifiers, an equivalence relation capturing the intuitive property that two identifiers are aliases if they denote the same abstract variable. This is somewhat reminiscent of the early work of Landin [19]. Technically, our semantics is *fully abstract* with respect to partial correctness behaviour. Full abstraction [23,28,29] 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 behaviour. Location-based semantics typically fail to be fully abstract, because semantically distinct terms can nevertheless induce precisely the same partial correctness behaviour in all program contexts. Moreover, in a location semantics, many different environment-store pairs represent the same sharing relation and assign equal values to all identifiers, and should therefore be regarded as equivalent; indeed, all terms have "equivalent" effects on equivalent environment-store pairs, in a precise sense.

Hoare's influential paper [16] proposed an axiomatic basis for programming languages. Hoare's paper gave an elegant 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. The appeal and influence of Hoare's work owes much to its use of syntax-directed proof rules and the simplicity of his assertion language. Many authors have tried to extend Hoare's ideas to cover more complicated and powerful programming constructs, and a good survey is provided in [1]. Existing proof rules for aliasing seem to be fairly complicated in form [4,5], and many proof rules for blocks beg the question by explicitly assuming that there is no possibility of aliasing [17]. 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), although we do not address these features in this paper either.

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, by inadequate use of semantic properties in designing the logic, or by an inappropriate choice of

assertion language (but see Clarke [6] for examples of constructs which are inherently difficult to treat). 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, as in [15]. And it often happens that some proof rules which appear to be obviously sound are still difficult to prove correct. Apt [1] and de Bakker [2] discuss some notable examples, and Apt also makes the point that the choice of a semantics is a decisive factor for the complexity of soundness and completeness proofs.

Almost every semantics used in the literature to support formal reasoning about partial correctness has been based on a location model. The second part of the paper develops a proof system based instead on a sharing relation semantics. By choosing a more appropriate level of abstraction in our semantics we believe that it becomes easier to reason about the semantics of programs, and we are able to build a very simple Hoare-style proof system for the language. The semantic structure guides us to a choice of assertion language and proof rules. And the semantics can be used in a straightforward (if somewhat tedious) way to establish the soundness of the proof system. Locations will not be needed as part of the assertion language, because they are not used in the semantics.

This paper only considers a very simple programming language with a small number of program constructs. We omit conditional commands and loops, for instance, although it is not difficult to incorporate them. 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. Although all programs in the language described in this paper terminate, so that there is no need to make a distinction between total and partial correctness here, we nevertheless use the term "partial correctness" throughout the paper, since it is very easy to extend our definitions and results to include conditionals and loops, and when this is done our results do indeed concern partial correctness. We make some remarks at the end of the paper concerning the extension of our techniques to programming languages containing various other features.

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 semantic description is given at this stage. Next we describe a more or less traditional location semantics, prove some useful properties of the semantics, and motivate our decision to choose a more abstract model. This leads to the introduction of sharing relations and valuations, and we construct a denotational semantics based upon them after showing that the semantics of terms is uniquely determined by their effect on sharing relations and valuations.

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 term (command, declaration, or expression) in a program context (of the appropriate type). We define a family of *behavioural equivalence* relations on terms, 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 these relations for expressions and for commands, but not for declarations. Instead for declarations, contextual equivalence coincides with an abstraction of semantic equivalence. This in fact shows that the abstract semantics is itself fully abstract.

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 relatively complete in the usual sense [7]. We give some examples to illustrate the use of the system, and we demonstrate that some of the proof rules for blocks in the literature can be derived in our system. In the final section of the paper we compare our work with that of other researchers, draw some conclusions and make some suggestions for future research. We provide an Appendix containing definitions and lemmas omitted from the paper, and which contains sketched proofs of some of the results mentioned there.

Notation.

Throughout the paper, we make use of common concepts pertaining to relations and functions. Thus, a relation R between two sets X and Y is a subset of the Cartesian product $X \times Y$. The domain of such a relation is $\mathrm{dom}(R) = \{x \in X \mid \exists y \in Y.(x,y) \in R\}$, and its range $\mathrm{rge}(R)$ is $\{y \in Y \mid \exists x \in X.(x,y) \in R\}$. A relation R is a *partial function* if for every x in its domain there is a unique y with $(x,y) \in R$. When R is a partial function we write $R : X \to Y$. A partial function is *total* if its domain is the whole of X. When $R \subseteq X \times Y$ and $R' \subseteq Y \times Z$ are relations, their *composition* $R' \circ R \subseteq X \times Z$ is defined

$$R' \circ R \;=\; \{(x,z) \mid \exists y.(x,y) \in R \;\&\; (y,z) \in R'\}.$$

The composition of two partial functions is again a partial function.

We also make use of *restriction* operations on relations. When $R \subseteq X \times Y$ and A is a set, we write $R{\downarrow}A$ for the relation obtained by restricting the domain of R to A; this is the relation on $A \times Y$ defined by

$$R{\downarrow}A \;=\; \{(x,y) \in R \mid x \in A\}.$$

We also use $R \Downarrow A$ for the relation obtained by restricting both domain and range of R to A, and $R|A$ for $R \Downarrow (\mathrm{dom}(R) - A)$:

$$\begin{aligned} R \Downarrow A \;&=\; \{(x,y) \in R \mid x \in A \;\&\; y \in A\}, \\ R|A \;&=\; \{(x,y) \in R \mid x \notin A \;\&\; y \notin A\}. \end{aligned}$$

We also use $R\backslash A$ for the relation obtained by restricting R's domain to $X - A$, i.e.,

$$R\backslash A \;=\; \{(x,y) \in R \mid x \notin A\}.$$

And R/A denotes the result of restricting the domain to $Y - A$:

$$R/A \;=\; \{(x,y) \in R \mid y \notin A\}.$$

When $f : X \to Y$ is a partial function, $x \in X$ and $y \in Y$, we write $f + [x \mapsto y]$ for the partial function with domain $\mathrm{dom}(f) \cup \{x\}$ agreeing with f except at x, which it maps to y. More generally, if $f : X \to Y$ and $f' : X' \to Y'$ are partial functions we write $f + f'$ for the partial function with domain $\mathrm{dom}(f) \cup \mathrm{dom}(f')$ agreeing with f' on $\mathrm{dom}(f')$ and with f on $\mathrm{dom}(f) - \mathrm{dom}(f')$.

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.

The abstract syntax for our language is described as follows.

Identifiers.

We assume that the syntax of identifiers 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.

Expressions.

We also assume given the syntax of expressions; the precise syntax is unimportant, except that an identifier is an expression, so is a numeral, and we allow simple arithmetical operations on expressions. Wherever our semantic development depends on an assumption about expressions we will make the assumption explicit. By isolating the important properties of expressions in this way, without being explicit about the syntax of expressions, we will prove results which are applicable to a wide variety of expression languages. We assume the usual notion of a *free* (occurrence of an) identifier in an expression, and we write $\mathrm{free}[\![E]\!]$ for the set of identifiers which occur free in an expression E. Trivially, $\mathrm{free}[\![I]\!] = \{I\}$. An expression having no free identifier occurrences is said to be *closed.*

Declarations.

For the syntax of declarations we specify:

$$\Delta ::= \ \mathbf{null} \mid \ \mathbf{new}\ I = E \mid \ \mathbf{alias}\ I_0 = I_1 \mid \ \Delta_0;\ \Delta_1.$$

We associate with a declaration Δ the following syntactic sets: $\mathrm{dec}[\![\Delta]\!]$, the set of *declared identifiers*; $\mathrm{free}[\![\Delta]\!]$, the set of free identifiers; and $\alpha[\![\Delta]\!] \subseteq \mathrm{dec}[\![\Delta]\!] \times \mathrm{free}[\![\Delta]\!]$, the set of *aliases* established by Δ, summarized as a relation between the declared and free identifiers of Δ. It will also be convenient to use the abbreviation $\mathrm{ids}[\![\Delta]\!]$ for $\mathrm{dec}[\![\Delta]\!] \cup \mathrm{free}[\![\Delta]\!]$, the set

of identifiers occurring (either free or bound) in Δ. Formally, we define:

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

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

$$
\begin{aligned}
\alpha[\![\Delta]\!] &\subseteq \mathrm{dec}[\![\Delta]\!] \times \mathrm{free}[\![\Delta]\!] \\
\alpha[\![\mathbf{null}]\!] &= \emptyset \\
\alpha[\![\mathbf{new}\ I = E]\!] &= \emptyset \\
\alpha[\![\mathbf{alias}\ I_0 = I_1]\!] &= \{(I_0, I_1)\} \\
\alpha[\![\Delta_0;\ \Delta_1]\!] &= (\alpha[\![\Delta_1]\!]/\mathrm{dec}[\![\Delta_0]\!]) \cup (\alpha[\![\Delta_0]\!]\backslash\mathrm{dec}[\![\Delta_1]\!]) \cup (\alpha[\![\Delta_1]\!] \circ \alpha[\![\Delta_0]\!]).
\end{aligned}
$$

The clause defining $\alpha[\![\Delta_0;\ \Delta_1]\!]$ illustrates our restriction notation, and is equivalent to the following:

$$
\begin{aligned}
\alpha[\![\Delta_0;\ \Delta_1]\!] &= \{(I', I) \in \alpha[\![\Delta_0]\!] \mid I' \notin \mathrm{dec}[\![\Delta_1]\!]\} \\
&\cup\ \{(I', I) \in \alpha[\![\Delta_1]\!] \mid I \notin \mathrm{dec}[\![\Delta_0]\!]\} \\
&\cup\ \{(I', I) \mid \exists I_0.\ (I', I_0) \in \alpha[\![\Delta_1]\!]\ \&\ (I_0, I) \in \alpha[\![\Delta_0]\!]\}.
\end{aligned}
$$

It is clear from the definitions that $\mathrm{ids}[\![\Delta_0;\ \Delta_1]\!] = \mathrm{ids}[\![\Delta_0]\!] \cup \mathrm{ids}[\![\Delta_1]\!]$. Note that $\mathrm{dec}[\![\Delta]\!]$ and $\mathrm{free}[\![\Delta]\!]$ need not be disjoint: an identifier may have a free occurrence and a bound occurrence in the same declaration, as in for example the declaration **new** $x = x + 1$. A declaration is *closed* if it has no free identifiers.

Commands.

The purpose of a command is to alter the values of variables: We use the following syntax:

$$\Gamma ::= \mathbf{skip} \mid I{:=}E \mid \Gamma_0;\ \Gamma_1 \mid \mathbf{begin}\ \Delta;\ \Gamma\ \mathbf{end}.$$

Again, identifiers may occur free in commands; a formal definition of the set of free identifiers of a command is given by:

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

Again, a command is closed if it has no free identifiers.

Programs.

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. Thus, a program contains no free identifier occurrences, and is therefore closed. This syntactic constraint is reasonable and is commonly imposed in practical programming languages.

Of course, this is a particularly simple form of programming language. We have omitted conditionals and loops, and we have a very simple structure for programs. In fact, the program structure has been chosen to correspond with a simple notion of partial correctness. The evaluation of a result expression is analogous to evaluating a "post-condition" on the values of identifiers after executing a command, although of course this is not a boolean post-condition. The connection with the usual formulation of partial correctness will be made later in the paper, after the relevant definitions have been introduced. It should be clear how to extend the notion of program to allow (for example) sequential composition at the program level, or more than one result expression, possibly to be evaluated at several points during program execution (perhaps by adding an *output* command to the language).

Informal Semantics.

Informally, we may explain the semantics of these constructs as follows. We are deliberately a little vague here; in particular, we do not define precisely what is meant by a "variable". It suffices for the moment to assume only that variables can be *named*, and that a variable possesses a value, can be initialized, and can be updated. As usual, even at this informal level of description, we are careful to distinguish between variables and the identifiers used to name them.

Declarations. The purpose of a declaration is to introduce a new set of variables and to bind them to declared identifiers:

- The **null** declaration has no effect.
- A *simple* declaration of the form

$$\textbf{new } I = E$$

introduces a new variable and binds the identifier I to it; the initial value of the variable is the current (declaration time) value of the expression E. The declared identifier I is not an alias for any other identifier.

- A *sharing* declaration of the form

$$\textbf{alias } I_0 = I_1$$

binds I_0 to the variable named by I_1: the effect of the declaration is to make I_0 an alias of 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 initializes the value of I_0 to the current value of 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 Δ_0; Δ_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. An alias for an identifier declared in Δ_0 but not again in Δ_1 is determined by Δ_0. For an identifier declared in Δ_1 there are two (mutually exclusive) possibilities for its alias, depending on whether or not it is declared as an alias for an identifier declared in Δ_0.

Commands.

Informally, we explain the intended semantics as follows:

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

- An *assignment* $I{:=}E$ updates the variable named by I, changing its value 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 Γ_0; Γ_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. An identifier in $\mathrm{dec}[\![\Delta]\!]$ is said to be *local* to the block, and an identifier in $\mathrm{free}[\![\Gamma]\!] - \mathrm{dec}[\![\Delta]\!]$ is *global.* Assignments made within the block may affect and be affected by the variables introduced in the declaration; but the scope of the declaration does not extend outside the block, so that variables bound to local identifiers of the block become inaccessible on block exit.

Programs.

The meaning of a program **begin** Δ; Γ **result** E **end** will be the value of the expression E in the environment formed by first performing the declaration Δ and then executing the command Γ. The fact that a program is closed will ensure that this value is uniquely determined and that during program execution all identifiers used in the program have been declared. Moreover, since all declarations initialize their declared identifiers, there will be no attempts to evaluate uninitialized variables during program execution, so that we need not worry about runtime errors in the semantic definitions.

Example declarations.

1. The declaration

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

is closed. Its effect is to introduce two new variables named x and y, with the variable named x initialized to the value 1 and y to 2. We have $\mathrm{dec}[\![\Delta_0]\!] = \{x, y\}, \mathrm{free}[\![\Delta_0]\!] = \emptyset$, and $\alpha[\![\Delta_0]\!] = \emptyset$. None of the declared identifiers is an alias for any other identifier.

2. On the other hand, the declaration

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

contains a free occurrence of x, and the value used to initialize y depends on the current (declaration-time) value of this free identifier. Here we have $\mathrm{dec}[\![\Delta_1]\!] = \{x, y\}$ and $\mathrm{free}[\![\Delta_1]\!] = \{x\}$, but still $\alpha[\![\Delta_1]\!] = \emptyset$.

3. The declaration

$$\Delta_2: \quad \textbf{alias } x = y; \textbf{ alias } y = z$$

introduces an alias x for the "old" y, and an alias y for the old z. We have $\text{free}(\Delta_2) = \{y, z\}, \text{dec}(\Delta_2) = \{x, y\}$ and $\alpha(\Delta_2) = \{(x, y), (y, z)\}$.

4. If we reverse the order of the previous example we get

$$\Delta_3: \quad \textbf{alias } y = z; \textbf{ alias } x = y,$$

with $\text{free}(\Delta_3) = \{z\}$ and $\alpha(\Delta_3) = \{(x, z), (y, z)\}$. Both of the declared identifiers are aliases for z.

5. To illustrate the effects of nested declarations of the same identifier, consider

$$\Delta_4: \quad \textbf{alias } x = y; \textbf{ alias } x = z.$$

Here we get a single declared identifier x, and it will be an alias for z (*not* also for y, unless z happens already to be an alias for y). We get $\text{free}(\Delta_4) = \{y, z\}, \text{dec}(\Delta_4) = \{x\}$ and $\alpha(\Delta_4) = \{(x, z)\}$.

6. For an example mixing the two forms of declaration, consider

$$\Delta_5: \quad \textbf{alias } x = a; \textbf{ new } y = x; \textbf{ alias } z = x; \textbf{ alias } x = b.$$

Here we have $\text{free}(\Delta_5) = \{a, b\}, \text{dec}(\Delta_5) = \{x, y, z\}$ and $\alpha(\Delta_5) = \{(x, b), (z, a)\}$.

Example commands.

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 the global identifier 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 global identifier x. ∎

4. For $i = 0 \ldots 4$ let Γ_i be the block **begin** Δ_i; $x{:=}x + 1$; $y{:=}\ y + 1$ **end**, where the Δ_i are as in the previous set of examples. Both Γ_0 and Γ_1 are closed, and have no effect when executed. Both Γ_2 and Γ_4 will first increment by 1 all aliases for (the global) y, and then increment by 1 the aliases of z. Γ_3 increases by 2 the values of all aliases of z.

3. Semantics.

Locations, Environments and Stores.

We first define a standard location based semantics for our language. We assume given a countably infinite set **Loc** of locations, so that we can ignore problems associated with finite store limitations and storage overflow. Although it would be easy to modify our model to treat storage overflow errors accurately, we will not consider this issue here in any detail. We are primarily interested in proving properties of programs that are true provided no storage overflow occurs during program execution. Since in our language every program uses only a finite amount of storage we are effectively supposing that every program is executed on a store large enough for it.

It will also always be the case during program execution that all identifiers used in a program possess a value; thus we will not need to deal with runtime errors owing to failure to initialize a variable. Instead, we build this property directly into the semantic definitions.

As indicated earlier, the purpose of locations is to serve as "variables"; a declaration such as **new** $x = 0$ is described as binding the identifier x to a "fresh" location distinct from all locations currently in use, and initializing the contents of this location to 0; in slightly more abstract terms, the declaration introduces a new variable, named x, different from all variables currently declared, and initializes its value to 0.

An *environment* is a finite partial function u from **Ide** to **Loc**, and a *store* is a finite partial function s from **Loc** to V. We will use U and S for the sets of environments and stores respectively. The domain of an environment, $\mathrm{dom}(u) = \{I | \exists l \in \mathbf{Loc}.\ u(I) = l\}$, consists of the identifiers for which there is a current declaration; the domain $\mathrm{dom}(s)$ of a store consists of the locations which are currently active or in use; this is often called the *area* of s. Locations in $\mathrm{rge}(u) = \{l \in \mathbf{Loc} | \exists I \in \mathbf{Ide}.u(I) = l\}$ are *accessible*, and the locations in $\mathrm{dom}(s) - \mathrm{rge}(u)$ are *inaccessible*. Similarly, for a set X of identifiers, the locations accessible from X are in $u(X) = \{l \in \mathbf{Loc} \mid \exists I \in X.\ u(I) = l\}$.

A pair $\langle u, s\rangle$ is *consistent* if and only if $\mathrm{dom}(s) \supseteq \mathrm{rge}(u)$. Thus, $\langle u, s\rangle$ is consistent if and only if all accessible locations are in use; this property will hold at all times during program execution. We refer to a consistent pair $\langle u, s\rangle$ as a *configuration*, and we let $C \subseteq U \times S$ be the set of configurations. In addition, we define for a set X of identifiers the set of configurations C_X, to be:

$$C_X \ = \ \{\langle u, s\rangle \mid \mathrm{dom}(u) \supseteq X \ \ \& \ \ \mathrm{dom}(s) \supseteq \mathrm{rge}(u)\}.$$

We extend the restriction notation to configurations in the following way:

$$\langle u, s\rangle{\downarrow}X \ = \ \langle u{\downarrow}X, s{\downarrow}u(X)\rangle.$$

We also extend the overwriting operation to configurations, defining

$$\langle u, s\rangle + \langle u', s'\rangle \ = \ \langle u + u', s + s'\rangle.$$

Let *new* be any function from finite subsets of **Loc** to **Loc** satisfying the following "newness" property:

$$\forall A \subseteq_{\mathrm{fin}} \mathbf{Loc}.(\mathrm{new}(A) \notin A),$$

i.e., a *new* function simply selects a location which is outside of the set supplied. Since every store has a finite domain, and we assumed that **Loc** is infinite, $\mathrm{new}(\mathrm{dom}(s))$ will always be defined. The consistency property of configurations guarantees that the location chosen by $\mathrm{new}(\mathrm{dom}(s))$ is inaccessible from u, so that no "accidental" aliasing will occur when storage is allocated.

Standard Semantics.

It is common to define semantic functions of the following types for a language such as ours:

$$\mathcal{E} : \mathbf{Exp} \to C \to V$$
$$\mathcal{M} : \mathbf{Com} \to C \to S$$
$$\mathcal{D} : \mathbf{Dec} \to C \to C$$
$$\mathcal{P} : \mathbf{Prog} \to V.$$

These types reflect the facts that commands change the store but not the environment (except locally during block execution), and that declarations affect the environment and also the store (since declarations claim storage and initialize their variables). However, these types are somewhat looser than necessary, since they do not build in and take advantage of certain semantic properties. For instance, $\mathcal{E}[\![E]\!]\langle u, s\rangle$ need only be defined when all of the free identifiers of E have proper values, *i.e.* when $\mathrm{dom}(u) \supseteq \mathrm{free}[\![E]\!]$, and similarly for declarations and commands. Moreover, for declarations it is enough to specify the result $\mathcal{D}[\![\Delta]\!]\langle u, s\rangle$ explicitly on $\mathrm{dec}(\Delta)$. It is also convenient for presentational purposes to modify the type of $\mathcal{M}$ so that the environment is carried along too, although as we remarked before no command changes the environment. Hence, we will actually define semantic functions satisfying the following type constraints:

$$\mathcal{E}[\![E]\!] : C_{\mathrm{free}(E)} \to V$$
$$\mathcal{D}[\![\Delta]\!] : C_{\mathrm{free}(\Delta)} \to C_{\mathrm{dec}(\Delta)}$$
$$\mathcal{M}[\![\Gamma]\!] : C_{\mathrm{free}(\Gamma)} \to C_{\mathrm{free}(\Gamma)}.$$

We omit details of the semantics of expressions, but we make the standard assumption that the value of an expression depends only on the values of its free identifiers. The clauses are fairly standard, and we refer to [10,22,34,37] for motivation.

$$\mathcal{E}[\![I]\!]\langle u, s\rangle \;=\; s(u(I))$$

$$\begin{aligned}
&\mathcal{D}[\![\mathbf{null}]\!]\langle u, s\rangle \;=\; \langle \emptyset, \emptyset\rangle \\
&\mathcal{D}[\![\mathbf{new}\ I = E]\!]\langle u, s\rangle \;=\; \langle [I \mapsto l], [l \mapsto \mathcal{E}[\![E]\!]\langle u, s\rangle]\rangle, \quad \text{where } l = \mathrm{new}(\mathrm{dom}(s)) \\
&\mathcal{D}[\![\mathbf{alias}\ I_0 = I_1]\!]\langle u, s\rangle \;=\; \langle [I_0 \mapsto u(I_1)], s{\downarrow}u(I_1)\rangle \\
&\mathcal{D}[\![\Delta_0;\ \Delta_1]\!]\langle u, s\rangle \;=\; \langle u_0 + u_1, s_0 + s_1\rangle \\
&\quad \text{where } \langle u_0, s_0\rangle \;=\; \mathcal{D}[\![\Delta_0]\!]\langle u, s\rangle \\
&\quad \text{and } \langle u_1, s_1\rangle \;=\; \mathcal{D}[\![\Delta_1]\!]\langle u + u_0, s + s_0\rangle
\end{aligned}$$

$$\begin{aligned}
&\mathcal{M}[\![\mathbf{skip}]\!]\langle u, s\rangle \;=\; \langle u, s\rangle \\
&\mathcal{M}[\![I{:=}E]\!]\langle u, s\rangle \;=\; \langle u, s + [u(I) \mapsto \mathcal{E}[\![E]\!]\langle u, s\rangle]\rangle \\
&\mathcal{M}[\![\Gamma_1; \Gamma_2]\!]\langle u, s\rangle \;=\; \mathcal{M}[\![\Gamma_2]\!](\mathcal{M}[\![\Gamma_1]\!]\langle u, s\rangle) \\
&\mathcal{M}[\![\mathbf{begin}\ \Delta;\ \Gamma\ \mathbf{end}]\!]\langle u, s\rangle \;=\; \langle u, s''{\downarrow}\mathrm{rge}(u)\rangle, \\
&\quad \text{where } \langle u', s'\rangle \;=\; \mathcal{D}[\![\Delta]\!]\langle u, s\rangle \\
&\quad \text{and } \langle u + u', s''\rangle = \mathcal{M}[\![\Gamma]\!]\langle u + u', s + s'\rangle
\end{aligned}$$

$$\mathcal{P}[\![\mathbf{begin}\ \Delta;\ \Gamma;\ \mathbf{result}\ E\ \mathbf{end}]\!] \;=\; \mathcal{E}[\![E]\!](\mathcal{M}[\![\Gamma]\!](\mathcal{D}[\![\Delta]\!]\langle \emptyset, \emptyset\rangle)).$$

Storage Management.

Several observations about the handling of storage in these definitions are worthy of note. Firstly, notice again the rôle played by the *new* function in the semantic clauses. We did not specify the exact description of this function, merely assuming that it satisfied the so-called newness property.

Secondly, the semantic clause for a block uses *restriction* to $\mathrm{rge}(u)$. We can reformulate this clause as follows:

$$\mathcal{M}[\![\textbf{begin } \Delta;\ \Gamma \textbf{ end}]\!]\langle u, s\rangle \ = \ \langle u, (\mathcal{S}^{\dagger}[\![\Gamma]\!](\mathcal{D}^{\dagger}[\![\Delta]\!]\langle u, s\rangle))\downarrow \mathrm{rge}(u)\rangle. \tag{1}$$

Our use of this operation models the de-allocation on block exit of all locations claimed by **new** declarations on block entry. In fact, to be precise, we have specified that the locations inaccessible from the current environment be de-allocated, and this will certainly include these newly claimed locations. This is, therefore, a semantics which can be implemented with an elementary type of "garbage collection".

Some alternative semantic clauses for blocks are also worth discussing. One is obtained by omitting the restriction, putting

$$\mathcal{M}[\![\textbf{begin } \Delta;\ \Gamma \textbf{ end}]\!]\langle u, s\rangle \ = \ \langle u, \mathcal{S}^{\dagger}[\![\Gamma]\!](\mathcal{D}^{\dagger}[\![\Delta]\!]\langle u, s\rangle)\rangle. \tag{2}$$

Under this interpretation, once a location is allocated it never becomes deallocated, suggesting an implementation without garbage collection. Another alternative is to restrict instead to $\mathrm{dom}(s)$, setting

$$\mathcal{M}[\![\textbf{begin } \Delta;\ \Gamma \textbf{ end}]\!]\langle u, s\rangle \ = \ \langle u, (\mathcal{S}^{\dagger}[\![\Gamma]\!](\mathcal{D}^{\dagger}[\![\Delta]\!]\langle u, s\rangle))\downarrow \mathrm{dom}(s)\rangle. \tag{3}$$

By restricting to $\mathrm{dom}(s)$ we de-allocate all locations claimed at the head of the block but leave open the possibility of inaccessible locations: once a location becomes inaccessible it remains so. In particular, if $\mathrm{rge}(u)$ is a proper subset of $\mathrm{dom}(s)$ then block exit will cause a shrinking of the size of the store.

Thirdly, the semantics given here does not avoid unnecessary allocation of locations when executing a declaration such as

$$\textbf{new } x = 0;\ \textbf{new } x = 1,$$

since according to the semantics above this causes the allocation of two locations, only one of which remains accessible. Of course, the inaccessible location will also be deallocated on block exit, but it remains in the domain of the store during block execution. It is primarily for this reason that we defined a configuration to be a pair $\langle u, s\rangle$ with $\mathrm{rge}(u) \subseteq \mathrm{dom}(s)$. It is possible to modify the semantics of blocks so that a more careful allocation strategy is used, say by additionally de-allocating inaccessible locations immediately before executing the body of a block. If we define $\langle u, s\rangle^* = \langle u, s\downarrow \mathrm{rge}(u)\rangle$ we may express this as:

$$\mathcal{M}[\![\textbf{begin } \Delta;\ \Gamma \textbf{ end}]\!]\langle u, s\rangle \ = \ \langle u, \mathcal{S}^{\dagger}[\![\Gamma]\!](\mathcal{D}^{\dagger}[\![\Delta]\!]\langle u, s\rangle)^*\rangle^*. \tag{5}$$

This would then enable us to modify the semantic model by employing configurations that enjoy an even tighter consistency property: that $\mathrm{dom}(s) = \mathrm{rge}(u)$. Another way to do this is to redefine the semantic function $\mathcal{D}$ by modifying the clause for sequential composition to:

$$\begin{aligned} \mathcal{D}[\![\Delta_0;\ \Delta_1]\!]\langle u, s\rangle \ &= \ \langle u_0 + u_1, s_0 + s_1\rangle^*, \\ \text{where } \langle u_0, s_0\rangle \ &= \ \mathcal{D}[\![\Delta_0]\!]\langle u, s\rangle \\ \text{and } \ \langle u_1, s_1\rangle \ &= \ \mathcal{D}[\![\Delta_1]\!]\langle u + u_0, s + s_0\rangle^*. \end{aligned}$$

From the point of view of implementation, however, it is questionable if garbage collection should be used after every block entry and every block exit.

We now have a variety of slightly different semantic definitions, employing more or less restricted forms of storage management. We chose to deallocate inaccessible locations on block exit, with the result that our semantics, as described by (1) above, satisfies:

$$\mathcal{M}[\![\textbf{begin } \Delta;\ \textbf{skip end}]\!]\langle u, s\rangle \ = \ \langle u, s{\downarrow}\mathrm{rge}(u)\rangle.$$

This is also true of the semantics defined with (4). But if we use definition (2) or (3) we get

$$\mathcal{M}[\![\textbf{begin } \Delta;\ \textbf{skip end}]\!]\langle u, s\rangle \ = \ \langle u, s\rangle.$$

Similar equations hold for blocks like **begin null**; Γ **end** with a trivial declaration.

None of these fine points of storage allocation are important from a point of view of proving correctness properties of programs, given our assumption that there is sufficient storage space to prevent overflow. Nor are the distinctions between $\langle u, s\rangle$ and $\langle u, s{\downarrow}\mathrm{rge}(u)\rangle$ relevant in this context. We will take care in the rest of the paper to state semantic properties that remain true whatever storage management decisions are built into the semantics. Where relevant we state precisely what properties depend crucially on storage details.

Auxiliary Semantic Functions.

It will be convenient to introduce some auxiliary semantic functions. We will use

$$\mathcal{D}^{\dagger}[\![\Delta]\!] \ : \ C_{\mathrm{free}(\Delta)} \to C_{\mathrm{ids}(\Delta)},$$

$$\text{defined by} \qquad \mathcal{D}^{\dagger}[\![\Delta]\!]\langle u, s\rangle \ = \ \langle u, s\rangle + \mathcal{D}[\![\Delta]\!]\langle u, s\rangle.$$

Clearly, $\mathcal{D}^{\dagger}$ has the following denotational description:

$$\mathcal{D}^{\dagger}[\![\textbf{null}]\!]\langle u, s\rangle \ = \ \langle u, s\rangle$$

$$\mathcal{D}^{\dagger}[\![\textbf{new } I = E]\!]\langle u, s\rangle \ = \ \langle u + [I \mapsto l], s + [l \mapsto \mathcal{E}[\![E]\!]\langle u, s\rangle\rangle \quad \text{where } l = \mathrm{new}(\mathrm{dom}(s))$$

$$\mathcal{D}^{\dagger}[\![\textbf{alias } I_0 = I_1]\!]\langle u, s\rangle \ = \ \langle u + [I_0 \mapsto u(I_1)], s\rangle$$

$$\mathcal{D}^{\dagger}[\![\Delta_0;\ \Delta_1]\!]\langle u, s\rangle \ = \ \mathcal{D}^{\dagger}[\![\Delta_1]\!](\mathcal{D}^{\dagger}[\![\Delta_0]\!]\langle u, s\rangle).$$

Each of $\mathcal{D}^{\dagger}$ and $\mathcal{D}$ can be defined in terms of the other, since we also have the obvious identity

$$\mathcal{D}[\![\Delta]\!]\langle u, s\rangle \ = \ (\mathcal{D}^{\dagger}[\![\Delta]\!]\langle u, s\rangle){\downarrow}\mathrm{dec}[\![\Delta]\!].$$

Using $\mathcal{D}^{\dagger}$ may simplify presentation of certain results. For instance, the semantic equation for blocks may be rewritten as

$$\mathcal{M}[\![\textbf{begin } \Delta;\ \Gamma \textbf{ end}]\!]\langle u, s\rangle \ = \ \mathcal{M}[\![\Gamma]\!](\mathcal{D}^{\dagger}[\![\Delta]\!]\langle u, s\rangle){\downarrow}u,$$

if we define $\langle u^{\dagger}, s^{\dagger}\rangle{\downarrow}u = \langle u, s^{\dagger}{\downarrow}\mathrm{rge}(u)\rangle$.

We also introduce auxiliary functions describing the effect of a declaration on the environment and on the store separately: $\mathcal{R}[\![\Delta]\!]\langle u, s\rangle$ and $\mathcal{S}[\![\Delta]\!]\langle u, s\rangle$ will be the environment and the store of $\mathcal{D}[\![\Delta]\!]\langle u, s\rangle$, and similarly for $\mathcal{R}^{\dagger}, \mathcal{S}^{\dagger}$, and $\mathcal{D}^{\dagger}$:

$$\mathcal{D}[\![\Delta]\!]\langle u, s\rangle \ = \langle \mathcal{R}[\![\Delta]\!]\langle u, s\rangle, \mathcal{S}[\![\Delta]\!]\langle u, s\rangle\rangle$$

$$\mathcal{D}^{\dagger}[\![\Delta]\!]\langle u, s\rangle \ = \langle \mathcal{R}^{\dagger}[\![\Delta]\!]\langle u, s\rangle, \mathcal{S}^{\dagger}[\![\Delta]\!]\langle u, s\rangle\rangle.$$

These auxiliary functions can also be described denotationally, i.e. by structural induction on Δ. The details are obvious, as are the relationships between them:

$$\begin{aligned}\mathcal{R}^\dagger[\![\Delta]\!]\langle u,s\rangle &= u + \mathcal{R}[\![\Delta]\!]\langle u,s\rangle,\\ \mathcal{S}^\dagger[\![\Delta]\!]\langle u,s\rangle &= s + \mathcal{S}[\![\Delta]\!]\langle u,s\rangle.\end{aligned}$$

It will also be useful to let $\mathcal{S}^\dagger[\![\Gamma]\!]\langle u,s\rangle$ denote the store of $\mathcal{M}[\![\Gamma]\!]\langle u,s\rangle$. From the definitions, it is clear that the environment of $\mathcal{M}[\![\Gamma]\!]\langle u,s\rangle$ is still u.

Finally, we define an auxiliary semantic function that describes precisely the aliasing relationships among global and local identifiers of a block:

$$\begin{aligned}\mathcal{T}[\![\Delta]\!]\langle u,s\rangle &\subseteq \mathrm{dom}(\mathcal{R}[\![\Delta]\!]\langle u,s\rangle) \times \mathrm{dom}(u),\\ \mathcal{T}[\![\Delta]\!]\langle u,s\rangle &= \{(I',I) \mid u(I) = \mathcal{R}[\![\Delta]\!]\langle u,s\rangle(I')\}.\end{aligned}$$

Similarly we may define

$$\begin{aligned}\mathcal{T}^\dagger[\![\Delta]\!]\langle u,s\rangle &\subseteq \mathrm{dom}(\mathcal{R}^\dagger[\![\Delta]\!]\langle u,s\rangle) \times \mathrm{dom}(u)\\ \mathcal{T}^\dagger[\![\Delta]\!]\langle u,s\rangle &= \{(I',I) \mid u(I) = \mathcal{R}^\dagger[\![\Delta]\!]\langle u,s\rangle(I')\}.\end{aligned}$$

Examples (revisited).

Let $\langle u,s\rangle$ be a configuration defined on x,y,z with $\mathrm{dom}(s) = \mathrm{rge}(u)$. Let $l_0 = \mathrm{new}(\mathrm{dom}(s))$ and $l_1 = \mathrm{new}(\mathrm{dom}(s) \cup \{l_0\})$. Let l_x, l_y, l_z be the locations $u(x), u(y), u(z)$. Let v_x, v_y, v_z be the values of x,y,z respectively in $\langle u,s\rangle$. Then we have

$$\begin{aligned}\mathcal{D}[\![\Delta_0]\!]\langle u,s\rangle &= \langle [x \mapsto l_0, y \mapsto l_1], [l_0 \mapsto 1, l_1 \mapsto 2]\rangle\\ \mathcal{D}[\![\Delta_1]\!]\langle u,s\rangle &= \langle [x \mapsto l_1, y \mapsto l_0], [l_1 \mapsto 1, l_0 \mapsto 2]\rangle\end{aligned}$$

If l_y and l_z are distinct, then

$$\begin{aligned}\mathcal{D}[\![\Delta_2]\!]\langle u,s\rangle &= \langle [x \mapsto l_y, y \mapsto l_z], [l_y \mapsto v_y, l_z \mapsto v_z]\rangle\\ \mathcal{D}[\![\Delta_3]\!]\langle u,s\rangle &= \langle [x \mapsto l_z, y \mapsto l_z], [l_z \mapsto v_z]\rangle\\ \mathcal{D}[\![\Delta_4]\!]\langle u,s\rangle &= \langle [x \mapsto l_z], [l_z \mapsto v_z]\rangle.\end{aligned}$$

We also have

$$\begin{aligned}\mathcal{M}[\![\Gamma_0]\!]\langle u,s\rangle &= \langle u,s\rangle\\ \mathcal{M}[\![\Gamma_1]\!]\langle u,s\rangle &= \langle u,s\rangle\\ \mathcal{M}[\![\Gamma_2]\!]\langle u,s\rangle &= \langle u, s + [l_y \mapsto (v_z+1), l_z \mapsto (v_z+1)]\rangle && \text{if } l_y \neq l_z\\ &= \langle u, s + [l_y \mapsto (v_y+2)]\rangle && \text{if } l_y = l_z.\end{aligned}$$

Semantic Properties.

Since identifiers are aliases when they denote the same location, it is natural to define, for a configuration $\langle u,s\rangle$, the *sharing relation* share(u) on dom(u), to be:

$$\mathrm{share}(u) = \{(I,I') \mid u(I) = u(I')\}.$$

This is clearly an equivalence relation on dom(u). For $I \in \mathrm{dom}(u)$ we call share$(u)(I) = \{I' \mid u(I') = u(I)\}$ the *sharing class* of I in u. We also define the ***valuation***

$$\mathrm{val}(u,s) = s \circ u,$$

which is clearly a total map from $\mathrm{dom}(u)$ to V giving *values* to the identifiers in $\mathrm{dom}(u)$. Obviously, by definition, $\mathrm{val}(u,s)$ gives the same value to all identifiers in the same sharing class.

We define a natural equivalence, called *sharing equivalence*, on environment-store pairs: $\langle u,s\rangle$ and $\langle u',s'\rangle$ are sharing equivalent iff $\mathrm{dom}(u)=\mathrm{dom}(u')$, both u and u' determine the same sharing relation, and $s\circ u$ and $s'\circ u'$ give the same values to all identifiers. We also define a relativized version: $\langle u,s\rangle$ and $\langle u',s'\rangle$ are sharing equivalent on a set X of identifiers if $\mathrm{dom}(u)\cap X = \mathrm{dom}(u')\cap X$, both u and u' determine the same sharing relationships among identifiers in X, and $s\circ u$ and $s'\circ u'$ give the same values to all identifiers in X. We write $\langle u,s\rangle \approx \langle u',s'\rangle$ to indicate sharing equivalence, and $\langle u,s\rangle \approx_X \langle u',s'\rangle$ to indicate the relativized version.

Definition. The relation $\approx$ (*sharing equivalence*) on configurations is defined by

$$\begin{aligned}\langle u_1,s_1\rangle \approx \langle u_2,s_2\rangle \quad \Leftrightarrow \quad & \mathrm{dom}(u_1)=\mathrm{dom}(u_2) \ \ \&\ \ \mathrm{share}(u_1)=\mathrm{share}(u_2)\\ & \&\ \ \mathrm{val}(u_1,s_1)=\mathrm{val}(u_2,s_2).\end{aligned}$$

Definition. The relativized version of sharing equivalence to a finite set of identifiers X is defined (for $\mathrm{dom}(u)=\mathrm{dom}(u')\supseteq X$) by:

$$\langle u,s\rangle \approx_X \langle u',s'\rangle \quad \Leftrightarrow \quad \mathrm{val}(u,s)\!\downarrow\! X = \mathrm{val}(u',s')\!\downarrow\! X \ \ \&\ \ \mathrm{share}(u)\Downarrow X = \mathrm{share}(u')\Downarrow X.$$

As usual, $\mathrm{val}(u,s)\!\downarrow\! X$ denotes the restriction to X of $\mathrm{val}(u,s)$ and $\mathrm{share}(u)\Downarrow X$ is the restriction of $\mathrm{share}(u)$ to a relation on $X\times X$:

$$\mathrm{share}(u)\Downarrow X = \{(I,I')\in \mathrm{share}(u) \mid I\in X \ \ \&\ \ I'\in X\}.$$

It is clear that this is in fact an equivalence relation on X.

Notice the obvious property that $\langle u,s\rangle \approx \langle u, s\!\downarrow\!\mathrm{rge}(u)\rangle$. Thus, in *all* of the various semantics discussed earlier, we get the relationships

$$\begin{aligned}\mathcal{M}[\![\textbf{begin } \Delta;\ \textbf{skip end}]\!]\langle u,s\rangle &\approx \langle u,s\rangle,\\ \mathcal{M}[\![\textbf{begin null};\ \Gamma\ \textbf{end}]\!]\langle u,s\rangle &\approx \langle u,s\rangle.\end{aligned}$$

In other words, $\approx$ blurs the distinctions between semantics which differ in inessential details of storage handling.

It is perhaps worth noticing that sharing equivalence of environment-store pairs coincides with an equivalence based on *permutation* of locations. If $\langle u,s\rangle$ and $\langle u',s'\rangle$ are sharing equivalent, then we may define a bijection π from $\mathrm{rge}(u)$ to $\mathrm{rge}(u')$ by setting $\pi(u(I))=u'(I)$ for all I in $\mathrm{dom}(u)=\mathrm{dom}(u')$. This is a bijection, since $\pi(u(I_1))=\pi(u(I_2))$ iff $u'(I_1)=u'(I_2)$ iff $u(I_1)=u(I_2)$ by assumption. The converse is also true: if π is a bijection from $\mathrm{rge}(u)$ and we define $u'=\pi\circ u$ and $s'=s\circ\pi^{-1}$ then $\langle u,s\rangle$ and $\langle u',s'\rangle$ are sharing equivalent. The bijection π behaves rather like a substitution of locations for locations. Although this gives us an alternative description of sharing equivalence, we will not use it in the paper.

Sharing equivalence, and the concepts of sharing relations and valuations, play an important role in formulating and proving semantic properties of our language.

Aliasing.

When we introduced the syntax of declarations we also defined the syntactic function $\alpha(\Delta) \subseteq \mathrm{dec}(\Delta) \times \mathrm{free}(\Delta)$ and said that it expresses the aliasing relationship between identifiers declared in Δ and the already declared identifiers occurring free in Δ. We now make this precise.

Lemma (i). For all Δ and all $\langle u, s\rangle \in C_{\mathrm{free}(\Delta)}$,

$$\mathcal{T}[\![\Delta]\!]\langle u, s\rangle \;=\; \alpha[\![\Delta]\!] \circ \mathrm{share}(u).$$

Proof. By structural induction on Δ, using the inductive definitions of $\mathcal{D}[\![\Delta]\!]$ and $\alpha[\![\Delta]\!]$.

By structural induction on Δ.

- The cases **null** and **new** $I = E$ are vacuously true, since in each of these cases $\alpha(\Delta) = \emptyset$ and $\mathcal{T}[\![\Delta]\!]\langle u, s\rangle = \emptyset$.
- For Δ of form **alias** $I_0 = I_1$ we have

$$\begin{aligned}
\alpha(\Delta) &= \{(I_0, I_1)\} \\
\mathcal{R}[\![\Delta]\!]\langle u, s\rangle &= \langle [I_0 \mapsto u(I_1)], s{\downarrow}u(I_1)\rangle \\
\mathcal{T}[\![\Delta]\!]\langle u, s\rangle &= \{(I_0, I) \mid u(I) = u(I_1)\} \;=\; \{(I_0, I) \mid (I_1, I) \in \mathrm{share}(u)\}
\end{aligned}$$

and the result holds.

- For $\Delta = \Delta_0; \Delta_1$, let $\langle u_0, s_0\rangle = \mathcal{D}[\![\Delta_0]\!]\langle u, s\rangle$ and $\langle u_1, s_1\rangle = \mathcal{D}[\![\Delta_1]\!]\langle u + u_0, s + s_0\rangle$, so that

$$\begin{aligned}
\mathcal{R}[\![\Delta]\!]\langle u, s\rangle &= u_0 + u_1 \\
\mathcal{T}[\![\Delta]\!]\langle u, s\rangle &= \{(I', I) \mid u(I) = (u_0 + u_1)(I')\}.
\end{aligned}$$

By the induction hypothesis for Δ_0 and for Δ_1,

$$\begin{aligned}
\mathcal{T}[\![\Delta_0]\!]\langle u, s\rangle &= \alpha(\Delta_0) \circ \mathrm{share}(u) \\
\mathcal{T}[\![\Delta_1]\!]\langle u + u_0, s + s_0\rangle &= \alpha(\Delta_1) \circ \mathrm{share}(u + u_0)
\end{aligned}$$

And by definition we also have, for $I' \in \mathrm{dec}(\Delta_0)$ and $I \in \mathrm{dom}(u)$,

$$\begin{aligned}
(I_0, I') \in \mathrm{share}(u + u_0) &\Leftrightarrow (I_0, I') \in \mathcal{T}[\![\Delta_0]\!]\langle u, s\rangle \\
&\Leftrightarrow (I_0, I') \in \alpha(\Delta_0) \circ \mathrm{share}(u).
\end{aligned}$$

Hence,

$$\begin{aligned}
\mathcal{T}[\![\Delta]\!]\langle u, s\rangle &= \{(I', I) \mid u(I) = (u_0 + u_1)(I')\} \\
&= \{(I', I) \mid I' \in \mathrm{dec}(\Delta_0) - \mathrm{dec}(\Delta_1) \;\&\; u(I) = u_0(I')\} \\
&\quad \cup \{(I', I) \mid I' \in \mathrm{dec}(\Delta_1) \;\&\; u(I) = u_1(I')\}.
\end{aligned}$$

Now $I' \in \mathrm{dec}(\Delta_0) - \mathrm{dec}(\Delta_1) \;\&\; u(I) = u_0(I')$ if and only if $(I', I) \in \alpha(\Delta_0) \circ \mathrm{share}(u)$ and $I' \notin \mathrm{dec}(\Delta_1)$, by the induction hypothesis for Δ_0.

And $u(I) = u_1(I')$ iff there is an $I_0 \in \mathrm{dom}(u) \cup \mathrm{dec}(\Delta_0)$ such that $u(I) = (u + u_0)(I_0) = u_1(I')$; equivalently, iff either $u(I) = u_0(I_0) = u_1(I')$ for some $I_0 \in \mathrm{dec}(\Delta_0)$ or $u(I) = u(I_0) = u_1(I')$ for some $I_0 \in \mathrm{dom}(u) - \mathrm{dec}(\Delta_0)$. The first of these situations is equivalent to $(I_0, I) \in \alpha(\Delta_0) \circ \mathrm{share}(u)$ and $(I', I_0) \in \alpha(\Delta_1) \circ \mathrm{share}(u + u_0)$ for some I_0, using the induction hypothesis for Δ_0 and for Δ_1. Rewriting, this is equivalent to

$$\exists I_0.\ (I', I_0) \in \alpha(\Delta_1) \;\&\; (I_0, I) \in \alpha(\Delta_0) \circ \mathrm{share}(u).$$

Similarly, the second of these situations is equivalent to

$$(I', I_0) \in \alpha(\Delta_1)/\mathrm{dec}(\Delta_0) \;\&\; (I, I_0) \in \mathrm{share}(u).$$

The result follows, since

$$\alpha(\Delta) \;=\; \alpha(\Delta_0)\backslash\mathrm{dec}(\Delta_1) \cup \alpha(\Delta_1)/\mathrm{dec}(\Delta_0) \cup (\alpha(\Delta_1) \circ \alpha(\Delta_0)).$$

∎

Invariance Properties.

We now state and prove some obvious *Invariance* properties of declarations and commands.

Proposition (ii). For all declarations Δ and all $\langle u, s\rangle \in C_{\text{free}(\Delta)}$,

(a). $\text{dom}(\mathcal{R}[\![\Delta]\!]\langle u, s\rangle) = \text{dec}(\Delta)$,

(b). $\text{rge}(\mathcal{R}[\![\Delta]\!]\langle u, s\rangle) \cap \text{dom}(s) \subseteq u(\text{free}(\Delta))$,

(c). $\mathcal{S}[\![\Delta]\!]\langle u, s\rangle$ agrees with s on $\text{dom}(\mathcal{S}[\![\Delta]\!]\langle u, s\rangle) \cap \text{dom}(s)$

Proof. An elementary structural induction on Δ.

- For $\Delta = \textbf{null}$ all three properties are trivially true, since $\mathcal{R}[\![\textbf{null}]\!]\langle u, s\rangle = \emptyset$, $\text{dec}[\![\textbf{null}]\!] = \emptyset$, and $\mathcal{S}[\![\textbf{null}]\!]\langle u, s\rangle = \emptyset$.
- For **new** $I = E$ we have

$$\begin{aligned}\mathcal{R}[\![\textbf{new } I = E]\!]\langle u, s\rangle &= [I \mapsto l]\\ \mathcal{S}[\![\textbf{new } I = E]\!]\langle u, s\rangle &= [l \mapsto \mathcal{E}[\![E]\!]\langle u, s\rangle]\\ &\text{where } l = \text{new}(\text{dom}(s)).\end{aligned}$$

The properties are obviously true, since l is not in $\text{dom}(s)$ and I is the only declared identifier.

- For **alias** $I_0 = I_1$ we have

$$\begin{aligned}\mathcal{R}[\![\textbf{alias } I_0 = I_1]\!]\langle u, s\rangle &= [I_0 \mapsto u(I_1)],\\ \mathcal{S}[\![\textbf{alias } I_0 = I_1]\!]\langle u, s\rangle &= [u(I_1) \mapsto s(u(I_1))].\end{aligned}$$

Properties (a) and (c) are obvious. Property (b) holds because I_1 is free in the declaration.

- For $\Delta = \Delta_0; \Delta_1$, we may assume the properties for Δ_0 and for Δ_1. We have

$$\begin{aligned}\mathcal{R}[\![\Delta]\!]\langle u, s\rangle &= u_0 + u_1,\\ \mathcal{S}[\![\Delta]\!]\langle u, s\rangle &= s_0 + s_1,\\ \text{where } \langle u_0, s_0\rangle &= \mathcal{R}[\![\Delta_0]\!]\langle u, s\rangle\\ \text{and } \langle u_1, s_1\rangle &= \mathcal{R}[\![\Delta_1]\!]\langle u + u_0, s + s_0\rangle.\end{aligned}$$

By induction hypothesis (a) for Δ_0 and Δ_1, $\text{dom}(u_0) = \text{dec}(\Delta_0)$ and $\text{dom}(u_1) = \text{dec}(\Delta_1)$. Hence, $\text{dom}(u_0 + u_1) = \text{dom}(u_0) \cup \text{dom}(u_1) = \text{dec}(\Delta_0) \cup \text{dec}(\Delta_1) = \text{dec}(\Delta_0;\ \Delta_1)$, as required for property (a).

By induction hypothesis (b) for Δ_0 and Δ_1,

$$\begin{aligned}\text{rge}(u_0) \cap \text{dom}(s) &\subseteq u(\text{free}(\Delta_0)),\\ \text{rge}(u_1) \cap \text{dom}(s + s_0) &\subseteq (u + u_0)(\text{free}(\Delta_1)).\end{aligned}$$

We also have, by definition of $+$, that $\text{dom}(s + s_0) = \text{dom}(s) \cup \text{dom}(s_0)$. Hence,

$$\begin{aligned}\text{rge}(\mathcal{R}[\![\Delta]\!]\langle u, s\rangle) \cap \text{dom}(s) &= \text{rge}(u_0 + u_1) \cap \text{dom}(s)\\ &= (\text{rge}(u_0) \cap \text{dom}(s)) \cup (\text{rge}(u_1) \cap \text{dom}(s))\\ &= (\text{rge}(u_0) \cap \text{dom}(s)) \cup ((\text{rge}(u_1) \cap \text{dom}(s + s_0)) \cap \text{dom}(s))\\ &\subseteq (\text{rge}(u_0) \cap \text{dom}(s)) \cup ((u + u_0)(\text{free}(\Delta_1)) \cap \text{dom}(s))\\ &\qquad\qquad \text{by (b) for } \Delta_1\\ &\subseteq (\text{rge}(u_0) \cap \text{dom}(s)) \cup u(\text{free}(\Delta_1) - \text{dec}(\Delta_0))\\ &\qquad\qquad \text{since } u_0(\text{free}(\Delta_1)) \subseteq \text{rge}(u_0)\\ &\subseteq u(\text{free}(\Delta_0)) \cup u(\text{free}(\Delta_1) - \text{dec}(\Delta_0)) \qquad \text{by (b) for } \Delta_0\\ &= u(\text{free}(\Delta))\end{aligned}$$

as required.

Finally, for property (c), we have the induction hypotheses:

$$s_0 \text{ agrees with } s \text{ on } \mathrm{dom}(s_0) \cap \mathrm{dom}(s);$$
$$s_1 \text{ agrees with } s + s_0 \text{ on } \mathrm{dom}(s_1) \cap \mathrm{dom}(s + s_0).$$

Hence, $s_0 + s_1$ agrees with s on $\mathrm{dom}(s_1) \cap \mathrm{dom}(s)$ and on $\mathrm{dom}(s_0) \cap \mathrm{dom}(s)$, as required. ∎

Corollary (iii). For all $\langle u, s\rangle \in C_{\mathrm{free}(\Delta)}$,

(a). $\mathrm{dom}(\mathcal{R}^\dagger[\![\Delta]\!]\langle u, s\rangle) = \mathrm{dom}(u) \cup \mathrm{dec}(\Delta)$.

(b). $\mathcal{R}^\dagger[\![\Delta]\!]\langle u, s\rangle$ agrees with u on $\mathrm{dom}(u) - \mathrm{dec}(\Delta)$.

(c). $\mathcal{S}^\dagger[\![\Delta]\!]\langle u, s\rangle$ agrees with s on $\mathrm{dom}(s)$.

Proof. Using Proposition (ii) and the definitions. ∎

Note also that it now follows from the definitions of $\mathcal{T}$ and $\mathcal{T}^\dagger$ that

$$\mathcal{T}[\![\Delta]\!]\langle u, s\rangle \subseteq \mathrm{dec}(\Delta) \times \mathrm{dom}(u),$$
$$\mathcal{T}^\dagger[\![\Delta]\!]\langle u, s\rangle \subseteq (\mathrm{dec}(\Delta) \cup \mathrm{dom}(u)) \times \mathrm{dom}(u).$$

We also get:

Corollary (*). For all Δ and all $\langle u, s\rangle \in C_{\mathrm{free}(\Delta)}$,

$$\mathrm{val}(u, s) = \mathrm{val}(u, s) + \mathrm{val}(\mathcal{D}^\dagger[\![\Delta]\!]\langle u, s\rangle) \circ \mathcal{T}^\dagger[\![\Delta]\!]\langle u, s\rangle.$$

Proof. It is enough to show that whenever $(I', I) \in \mathcal{T}^\dagger[\![\Delta]\!]\langle u, s\rangle$ it follows that $\mathrm{val}(u, s)(I) = \mathrm{val}(\mathcal{D}^\dagger[\![\Delta]\!]\langle u, s\rangle)(I')$. This is easy. When $(I', I) \in \mathcal{T}^\dagger[\![\Delta]\!]\langle u, s\rangle$ we have $(\mathcal{R}^\dagger[\![\Delta]\!]\langle u, s\rangle)(I') = u(I)$ by definition of $\mathcal{T}^\dagger$. This location is also in $\mathrm{dom}(s)$, by assumption on $\langle u, s\rangle$. By Proposition (ii), $\mathcal{S}^\dagger[\![\Delta]\!]\langle u, s\rangle$ agrees with s on this location. Thus,

$$\begin{aligned}
\mathrm{val}(\mathcal{D}^\dagger[\![\Delta]\!]\langle u, s\rangle)(I') &= (\mathcal{S}^\dagger[\![\Delta]\!]\langle u, s\rangle)(\mathcal{R}^\dagger[\![\Delta]\!]\langle u, s\rangle)(I')) \\
&= \mathcal{S}^\dagger[\![\Delta]\!]\langle u, s\rangle)(u(I)) \\
&= s(u(I)) \\
&= \mathrm{val}(u, s)(I)
\end{aligned}$$

as required. ∎

Corollary (iv). For all Δ and all $\langle u, s\rangle \in C_{\mathrm{free}(\Delta)}$,

$$\mathcal{D}^\dagger[\![\Delta]\!]\langle u, s\rangle \approx \langle u, s\rangle \quad \text{on } \mathrm{dom}(u) - \mathrm{dec}(\Delta).$$

Proof. Immediate from the previous Corollary and the definitions. ∎

Corollary (v). For any declaration Δ, and any set X of identifiers,

$$\mathcal{R}^\dagger[\![\Delta]\!]\langle u,s\rangle(X) \cap \mathrm{dom}(s) \subseteq u(X - \mathrm{dec}(\Delta)) \cup u(\mathrm{free}(\Delta)).$$

Proof. Using property (b) of Proposition (i), as in the proof of that proposition. $\mathcal{R}^\dagger[\![\Delta]\!]\langle u,s\rangle$ agrees with u on $X - \mathrm{dec}(\Delta)$, and agrees with $\mathcal{R}[\![\Delta]\!]\langle u,s\rangle$ on $X \cap \mathrm{dec}(\Delta)$. Thus,

$$\begin{aligned}(\mathcal{R}^\dagger[\![\Delta]\!]\langle u,s\rangle)(X) \cap \mathrm{dom}(s) &= u(X - \mathrm{dec}(\Delta)) \cup ((\mathcal{R}[\![\Delta]\!]\langle u,s\rangle)(X \cap \mathrm{dec}(\Delta)) \cap \mathrm{dom}(s)) \\ &\subseteq u(X - \mathrm{dec}(\Delta)) \cup (\mathrm{rge}(\mathcal{R}[\![\Delta]\!]\langle u,s\rangle) \cap \mathrm{dom}(s)) \\ &\subseteq u(X - \mathrm{dec}(\Delta)) \cup u(\mathrm{free}(\Delta)),\end{aligned}$$

as required. ∎

Proposition (vi). For all Γ and all $\langle u,s\rangle$ in $C_{\mathrm{free}(\Gamma)}$ the store $\mathcal{S}^\dagger[\![\Gamma]\!]\langle u,s\rangle$ agrees with s on $\mathrm{rge}(u) - u(\mathrm{free}(\Gamma))$.

Proof. By structural induction on Γ.

- For **skip** the result is trivial.
- For $I{:=}E$, the result follows from the fact that, for any v, the store $s + [u(I) \mapsto v]$ agrees with s except at $u(I)$.
- For Γ of form $\Gamma_0;\ \Gamma_1$, we have

$$\begin{aligned}\mathcal{M}[\![\Gamma]\!]\langle u,s\rangle &= \mathcal{M}[\![\Gamma_1]\!](\mathcal{M}[\![\Gamma_0]\!]\langle u,s\rangle), \\ \text{so that } \mathcal{S}^\dagger[\![\Gamma]\!]\langle u,s\rangle &= \mathcal{S}^\dagger[\![\Gamma_1]\!](\mathcal{M}[\![\Gamma_0]\!]\langle u,s\rangle).\end{aligned}$$

The environment of $\mathcal{M}[\![\Gamma_0]\!]\langle u,s\rangle$ is still u. Hence, by the induction hypothesis for Γ_1, $\mathcal{S}^\dagger[\![\Gamma_1]\!](\mathcal{M}[\![\Gamma_0]\!]\langle u,s\rangle)$ agrees with $\mathcal{S}^\dagger[\![\Gamma_0]\!]\langle u,s\rangle$ except on $u(\mathrm{free}(\Gamma_1))$. By the induction hypothesis for Γ_0, $\mathcal{S}^\dagger[\![\Gamma_0]\!]\langle u,s\rangle$ agrees with s except on $u(\mathrm{free}(\Gamma_0))$. Hence, $\mathcal{S}^\dagger[\![\Gamma]\!]\langle u,s\rangle$ agrees with s except on $u(\mathrm{free}(\Gamma_0)) \cup u(\mathrm{free}(\Gamma_1))$. Since $\mathrm{free}(\Gamma) = \mathrm{free}(\Gamma_0) \cup \mathrm{free}(\Gamma_1)$, the result follows.

- For a block Γ = **begin** $\Delta;\ \Gamma'$ **end**, we have

$$\mathcal{S}^\dagger[\![\Gamma]\!]\langle u,s\rangle = \mathcal{S}^\dagger[\![\Gamma']\!](\mathcal{D}^\dagger[\![\Delta]\!]\langle u,s\rangle).$$

We also have, by Corollary (v),

$$(\mathcal{R}^\dagger[\![\Delta]\!]\langle u,s\rangle)(\mathrm{free}(\Gamma')) \cap \mathrm{dom}(s) \subseteq u(\mathrm{free}(\Gamma') - \mathrm{dec}(\Delta)) \cup u(\mathrm{free}(\Delta)) = u(\mathrm{free}(\Gamma)),$$

By the inductive hypothesis for Γ', $\mathcal{S}^\dagger[\![\Gamma']\!](\mathcal{D}^\dagger[\![\Delta]\!]\langle u,s\rangle)$ agrees with $\mathcal{S}^\dagger[\![\Delta]\!]\langle u,s\rangle$ except on $\mathcal{R}^\dagger[\![\Delta]\!]\langle u,s\rangle(\mathrm{free}(\Gamma'))$. Hence, this agreement holds on $\mathrm{rge}(u) - u(\mathrm{free}(\Gamma))$. By Corollary (iii), $\mathcal{S}^\dagger[\![\Delta]\!]\langle u,s\rangle$ agrees with s on all of $\mathrm{rge}(u)$. Hence $\mathcal{S}^\dagger[\![\Gamma']\!](\mathcal{D}^\dagger[\![\Delta]\!]\langle u,s\rangle)$ agrees with s on $\mathrm{rge}(u) - u(\mathrm{free}(\Gamma))$ as required. ∎

Corollary. For all Γ and all $\langle u,s\rangle \in C_{\mathrm{free}(\Gamma)}$,

$$\mathcal{M}[\![\Gamma]\!]\langle u,s\rangle \approx \langle u,s\rangle \quad \text{on } \mathrm{dom}(u) - \mathrm{share}(u)(\mathrm{free}(\Gamma)). \quad ∎$$

Note also that

Corollary. For all Γ, Δ, and $\langle u, s\rangle \in C_{\text{free}(\textbf{begin } \Delta;\ \Gamma \textbf{ end})}$,

$$\text{val}(\mathcal{M}[\![\textbf{begin } \Delta;\ \Gamma \textbf{ end}]\!]\langle u, s\rangle) \ = \ \text{val}(u, s) + \text{val}(\mathcal{M}[\![\Gamma]\!](\mathcal{D}^{\dagger}[\![\Delta]\!]\langle u, s\rangle)) \circ \mathcal{T}^{\dagger}[\![\Delta]\!]\langle u, s\rangle.$$

Proof. As in the proof of the analogous property for declarations (Corollary (*)). It is enough to show that when I is not in the range of $\mathcal{T}^{\dagger}[\![\Delta]\!]\langle u, s\rangle$, the value in location $u(I)$ is unaffected by the block, and when $(I', I) \in \mathcal{T}^{\dagger}[\![\Delta]\!]\langle u, s\rangle$ we get

$$\text{val}(\mathcal{M}[\![\textbf{begin } \Delta;\ \Gamma \textbf{ end}]\!]\langle u, s\rangle(I) = \text{val}(\mathcal{M}[\![\Gamma]\!](\mathcal{D}^{\dagger}[\![\Delta]\!]\langle u, s\rangle))(I')$$

. The first part is easy: if I is not in the range of $\mathcal{T}^{\dagger}[\![\Delta]\!]\langle u, s\rangle$ then $u(I)$ is inaccessible from the environment $\mathcal{R}^{\dagger}[\![\Delta]\!]\langle u, s\rangle$. Hence, its contents cannot be modified by any command executing in that environment. For the second part, we argue as follows.

$$(I', I) \in \mathcal{T}^{\dagger}[\![\Delta]\!]\langle u, s\rangle \ \Leftrightarrow \ (\mathcal{R}^{\dagger}[\![\Delta]\!]\langle u, s\rangle)(I') \ = \ u(I).$$

Hence, if $(I', I) \in \mathcal{T}^{\dagger}[\![\Delta]\!]\langle u, s\rangle$ then

$$\begin{aligned}
\text{val}(\mathcal{M}[\![\textbf{begin } \Delta;\ \Gamma \textbf{ end}]\!]\langle u, s\rangle)(I) \ &= \ (\mathcal{S}^{\dagger}[\![\Gamma]\!](\mathcal{D}^{\dagger}[\![\Delta]\!]\langle u, s\rangle))(u(I)) \\
&= \ (\mathcal{S}^{\dagger}[\![\Gamma]\!](\mathcal{D}^{\dagger}[\![\Delta]\!]\langle u, s\rangle)(\mathcal{R}^{\dagger}[\![\Delta]\!]\langle u, s\rangle(I')) \\
&= \ \text{val}(\mathcal{M}^{\dagger}[\![\Gamma]\!](\mathcal{D}^{\dagger}[\![\Delta]\!]\langle u, s\rangle))(I')
\end{aligned}$$

∎

Influence Properties.

In statically scoped languages such as ours it is well known that the meaning of a term depends only on the semantic attributes of its free identifiers. One might attempt a formal statement of this type of property for commands as follows: if $\langle u, s\rangle$ and $\langle u, s'\rangle$ are configurations involving the same environment u, whose stores s and s' agree on the contents of the locations accessible from the identifiers occurring free in Γ, then the final stores $\mathcal{M}[\![\Gamma]\!]\langle u, s\rangle$ and $\mathcal{M}[\![\Gamma]\!]\langle u, s'\rangle$ will also agree on these locations.

However, in trying to *prove* these properties by structural induction on Γ the case for blocks requires an analogous property for declarations. But whereas $\mathcal{D}^{\dagger}[\![\Delta]\!]\langle u, s\rangle$ *extends* $\langle u, s\rangle$ in a natural manner, there is no need for it to have the same environment as $\mathcal{D}^{\dagger}[\![\Delta]\!]\langle u, s'\rangle$, given only that s and s' agree on $u(\text{free}(\Delta))$. Certainly their environments have the same domain, but the environments need not agree on $\text{dec}(\Delta)$, because the locations generated by $\text{new}(s)$ and $\text{new}(s')$ need not be the same. Similarly, the stores may differ.

Instead, the most one can say here is that $\mathcal{D}^{\dagger}[\![\Delta]\!]\langle u, s\rangle$ and $\mathcal{D}^{\dagger}[\![\Delta]\!]\langle u, s'\rangle$ are *sharing equivalent* as defined earlier. In fact, an even more general property holds: whenever $\langle u, s\rangle$ and $\langle u', s'\rangle$ are sharing equivalent on the free identifiers of Δ, $\mathcal{D}[\![\Delta]\!]\langle u, s\rangle$ and $\mathcal{D}[\![\Delta]\!]\langle u', s'\rangle$ are sharing equivalent on $\text{dec}(\Delta)$; and $\mathcal{D}^{\dagger}[\![\Delta]\!]\langle u, s\rangle$ is sharing equivalent to $\mathcal{D}^{\dagger}[\![\Delta]\!]\langle u', s'\rangle$ on $\text{ids}(\Delta)$.

In formulating Influence Properties we start with a natural assumption about the semantics of expressions.

Assumption 1. For all expressions E and all $\langle u, s\rangle$ and $\langle u', s'\rangle$ in $C_{\text{free}(E)}$,

$$\langle u, s\rangle \approx_{\text{free}(E)} \langle u', s'\rangle \quad \Rightarrow \quad \mathcal{E}[\![E]\!]\langle u, s\rangle = \mathcal{E}[\![E]\!]\langle u', s'\rangle. \quad ∎$$

Proposition 2. For all Δ and all $\langle u, s\rangle$ and $\langle u', s'\rangle$ in $C_{\text{free}(\Delta)}$,

$$\langle u, s\rangle \approx_X \langle u', s'\rangle \ \&\ X \supseteq \text{free}(\Delta) \quad \Rightarrow \quad \mathcal{D}^\dagger[\![\Delta]\!]\langle u, s\rangle \approx_{X \cup \text{dec}(\Delta)} \mathcal{D}^\dagger[\![\Delta]\!]\langle u', s'\rangle.$$

Proof. By structural induction on Δ.

- For $\Delta = \textbf{null}$ the result is trivially true.
- For Δ of form **new** $I = E$, assume $\langle u, s\rangle \approx_X \langle u', s'\rangle$ and $X \supseteq \text{free}(E)$. Let $l = \text{new}(\text{dom}(s))$ and $l' = \text{new}(\text{dom}(s'))$. By assumption on E, $\mathcal{E}[\![E]\!]\langle u, s\rangle = \mathcal{E}[\![E]\!]\langle u', s'\rangle = v$, say. Thus,

$$\begin{aligned}\mathcal{D}^\dagger[\![\Delta]\!]\langle u, s\rangle &= \langle u + [I \mapsto l], s + [l \mapsto v]\rangle,\\ \mathcal{D}^\dagger[\![\Delta]\!]\langle u', s'\rangle &= \langle u' + [I \mapsto l'], s' + [l' \mapsto v]\rangle.\end{aligned}$$

Since $l \notin \text{dom}(s)$ we get

$$\begin{aligned}\text{share}(u + [I \mapsto l]) &= (\text{share}(u)|\{I\}) \cup \{(I, I)\},\\ \text{val}(u + [I \mapsto l], s + [l \mapsto v]) &= \text{val}(u, s) + [I \mapsto v],\end{aligned}$$

so that

$$\begin{aligned}\text{share}(u + [I \mapsto l]) \Downarrow (X \cup \{I\}) &= (\text{share}(u) \Downarrow (X - \{I\})) \cup \{(I, I)\},\\ \text{val}(u + [I \mapsto l], s + [l \mapsto v])\downarrow(X \cup \{I\}) &= \text{val}(u, s)\downarrow X + [I \mapsto v].\end{aligned}$$

Similarly

$$\begin{aligned}\text{share}(u' + [I \mapsto l'])\downarrow(X \cup \{I\}) &= (\text{share}(u') \Downarrow (X - \{I\})) \cup \{(I, I)\},\\ \text{val}(u' + [I \mapsto l'], s' + [l' \mapsto v])\downarrow(X \cup \{I\}) &= \text{val}(u', s')\downarrow X + [I \mapsto v].\end{aligned}$$

By assumption, $\text{share}(u) \Downarrow X = \text{share}(u') \Downarrow X$ and $\text{val}(u, s)\downarrow X = \text{val}(u', s')\downarrow X$. The result follows immediately.

For Δ of form **alias** $I_0 = I_1$ we argue as follows. Assume $\langle u, s\rangle \approx_X \langle u', s'\rangle$ and $I_1 \in X$. We have

$$\begin{aligned}\mathcal{D}^\dagger[\![\Delta]\!]\langle u, s\rangle &= \langle u + [I_0 \mapsto u(I_1)], s\rangle,\\ \mathcal{D}^\dagger[\![\Delta]\!]\langle u', s'\rangle &= \langle u' + [I_0 \mapsto u'(I_1)], s'\rangle.\end{aligned}$$

Clearly,

$$\begin{aligned}\text{share}(u + [I_0 \mapsto u(I_1)]) &= (\text{share}(u)|\{I_0\}) \cup \{(I_0, I_0)\}\\ &\quad \cup \{(I, I_0), (I_0, I) \mid u(I) = u(I_1)\ \&\ I \neq I_0\},\\ \text{val}(u + [I_0 \mapsto u(I_1)], s) &= \text{val}(u, s) + [I_0 \mapsto s(u(I_1))].\end{aligned}$$

Similar equations hold for the primed versions. As in the previous case, the hypothesis that $\text{share}(u) \Downarrow X = \text{share}(u') \Downarrow X$ and that $\text{val}(u, s)\downarrow X = \text{val}(u', s')\downarrow X$ guarantees the desired property, that

$$\begin{aligned}\text{share}(u + [I_0 \mapsto u(I_1)]) \Downarrow (X \cup \{I_0\}) &= \text{share}(u' + [I_0 \mapsto u'(I_1)]) \Downarrow (X \cup \{I_0\})\\ \text{val}(u + [I_0 \mapsto u(I_1)], s)\downarrow(X \cup \{I_0\}) &= \text{val}(u' + [I_0 \mapsto u'(I_1)], s')\downarrow(X \cup \{I_0\}).\end{aligned}$$

For the case of $\Delta_0;\ \Delta_1$ we may assume the property holds of Δ_0 and of Δ_1. Assume $\langle u, s\rangle \approx_X \langle u', s'\rangle$ and $X \supseteq \text{free}(\Delta_0;\ \Delta_1)$. That is, $X \supseteq \text{free}(\Delta_0)$ and $X \supseteq (\text{free}(\Delta_1) - \text{dec}(\Delta_0)$. By definition,

$$\begin{aligned}\mathcal{D}^\dagger[\![\Delta_0;\ \Delta_1]\!]\langle u, s\rangle &= \mathcal{D}^\dagger[\![\Delta_1]\!](\mathcal{D}^\dagger[\![\Delta_0]\!]\langle u, s\rangle)\\ \mathcal{D}^\dagger[\![\Delta_0;\ \Delta_1]\!]\langle u', s'\rangle &= \mathcal{D}^\dagger[\![\Delta_1]\!](\mathcal{D}^\dagger[\![\Delta_0]\!]\langle u', s'\rangle).\end{aligned}$$

By the induction hypothesis for Δ_0, since $X \supseteq \text{free}(\Delta_0)$,

$$\mathcal{D}^\dagger[\![\Delta_0]\!]\langle u,s\rangle \approx_{X\cup\text{dec}(\Delta_0)} \mathcal{D}^\dagger[\![\Delta_0]\!]\langle u',s'\rangle.$$

By the induction hypothesis for Δ_1, since $X \cup \text{dec}(\Delta_0) \supseteq \text{free}(\Delta_1)$,

$$\mathcal{D}^\dagger[\![\Delta_1]\!](\mathcal{D}^\dagger[\![\Delta_0]\!]\langle u,s\rangle) \approx_{X\cup\text{dec}(\Delta_0)\cup\text{dec}(\Delta_1)} \mathcal{D}^\dagger[\![\Delta_1]\!](\mathcal{D}^\dagger[\![\Delta_0]\!]\langle u',s'\rangle).$$

Since $\text{dec}(\Delta_0;\ \Delta_1) = \text{dec}(\Delta_0) \cup \text{dec}(\Delta_1)$ the result follows.

∎

Corollary. For all Δ, and all $\langle u,s\rangle$ and $\langle u',s'\rangle$ in $C_{\text{free}(\Delta)}$,

$$\langle u,s\rangle \approx_{\text{free}(\Delta)} \langle u',s'\rangle \quad\Rightarrow\quad \mathcal{D}[\![\Delta]\!]\langle u,s\rangle \approx_{\text{dec}(\Delta)} \mathcal{D}[\![\Delta]\!]\langle u',s'\rangle.$$

Proof. Assume $\langle u,s\rangle \approx_{\text{free}(\Delta)} \langle u',s'\rangle$. Then by the previous result $\mathcal{D}^\dagger[\![\Delta]\!]\langle u,s\rangle \approx_{\text{ids}(\Delta)} \mathcal{D}^\dagger[\![\Delta]\!]\langle u',s'\rangle$. By definition, $\mathcal{R}^\dagger[\![\Delta]\!]\langle u,s\rangle$ agrees with $\mathcal{R}[\![\Delta]\!]\langle u,s\rangle$ on $\text{dec}(\Delta)$; and $\mathcal{S}^\dagger[\![\Delta]\!]\langle u,s\rangle$ agrees with $\mathcal{S}[\![\Delta]\!]\langle u,s\rangle$ on $\text{rge}(\mathcal{R}[\![\Delta]\!]\langle u,s\rangle)$. Thus, on identifiers in $\text{dec}(\Delta)$, both $\mathcal{D}^\dagger[\![\Delta]\!]\langle u,s\rangle$ and $\mathcal{D}[\![\Delta]\!]\langle u,s\rangle$ determine the same sharing relationships and the same valuation. Similarly for the primed version. The result follows immediately. ∎

The fact that the effect of a command depends only on the attributes of its free identifiers (specifically, their sharing properties and their values) is shown by:

Proposition 4. For all commands Γ and all $\langle u,s\rangle$ and $\langle u',s'\rangle$ in $C_{\text{free}(\Gamma)}$,

$$\langle u,s\rangle \approx_X \langle u',s'\rangle \ \&\ X \supseteq \text{free}(\Gamma) \quad\Rightarrow\quad \mathcal{M}[\![\Gamma]\!]\langle u,s\rangle \approx_{\text{free}(\Gamma)} \mathcal{M}[\![\Gamma]\!]\langle u',s'\rangle.$$

Proof. By structural induction on Γ.

It is enough to show that

$$\langle u,s\rangle \approx_X \langle u',s'\rangle \ \&\ X \supseteq \text{free}(\Gamma) \quad\Rightarrow\quad \text{val}(\mathcal{M}[\![\Gamma]\!]\langle u,s\rangle)\downarrow X \ = \ \text{val}(\mathcal{M}[\![\Gamma]\!]\langle u',s'\rangle)\downarrow X,$$

since the sharing property is trivial.

- For $\Gamma = \textbf{skip}$ this is obviously true.
- For $I{:=}E$ we have

$$\begin{aligned}\mathcal{M}[\![I{:=}E]\!]\langle u,s\rangle &= \langle u, s + [u(I) \mapsto \mathcal{E}[\![E]\!]\langle u,s\rangle]\rangle,\\ \text{so that } \text{val}(\mathcal{M}[\![I{:=}E]\!]\langle u,s\rangle) &= \text{val}(u,s) + [\text{share}(u)(I) \mapsto \mathcal{E}[\![E]\!]\langle u,s\rangle].\end{aligned}$$

Similarly, $\text{val}(\mathcal{M}[\![I{:=}E]\!]\langle u',s'\rangle) = \text{val}(u',s') + [\text{share}(u')(I) \mapsto \mathcal{E}[\![E]\!]\langle u',s'\rangle]$. Assuming that $\langle u,s\rangle \approx_X \langle u',s'\rangle$ and $X \supseteq \text{free}(E) \cup \{I\}$, we have

$$\begin{aligned}\text{share}(u)(I) \cap X &= \text{share}(u')(I) \cap X,\\ \text{val}(u,s)\downarrow X &= \text{val}(u',s')\downarrow X,\end{aligned}$$

from which our desired result follows.

- For Γ of form $\Gamma_0;\ \Gamma_1$ we argue as follows. Assume $\langle u,s\rangle \approx_X \langle u',s'\rangle$ and $X \supseteq \text{free}(\Gamma_0;\ \Gamma_1) = \text{free}(\Gamma_0) \cup \text{free}(\Gamma_1)$. By definition,

$$\mathcal{M}[\![\Gamma_0;\ \Gamma_1]\!]\langle u,s\rangle \ = \ \mathcal{M}[\![\Gamma_1]\!](\mathcal{M}[\![\Gamma_0]\!]\langle u,s\rangle).$$

By assumption, $X \supseteq \text{free}(\Gamma_0)$ and $X \supseteq \text{free}(\Gamma_1)$. Hence, by the induction hypothesis for Γ_0,

$$\mathcal{M}[\![\Gamma_0]\!]\langle u, s\rangle \approx_X \mathcal{M}[\![\Gamma_0]\!]\langle u', s'\rangle.$$

By the induction hypothesis for Δ_1,

$$\mathcal{M}[\![\Gamma_1]\!](\mathcal{M}[\![\Gamma_0]\!]\langle u, s\rangle \approx_X \mathcal{M}[\![\Gamma_1]\!](\mathcal{M}[\![\Gamma_0]\!]\langle u', s'\rangle),$$

as required.

- For a block of the form **begin** Δ; Γ' **end**, assume that $\langle u, s\rangle \approx_X \langle u', s'\rangle$ and $X \supseteq \text{free}(\Delta) \cup (\text{free}(\Gamma') - \text{dec}(\Delta))$. By Corollary (*),

$$\begin{aligned}
\text{val}(\mathcal{M}[\![\textbf{begin } \Delta;\ \Gamma' \textbf{ end}]\!]\langle u, s\rangle) &= \text{val}(\mathcal{M}[\![\Gamma']\!](\mathcal{D}^\dagger[\![\Delta]\!]\langle u, s\rangle)) \circ \mathcal{T}^\dagger[\![\Delta]\!]\langle u, s\rangle, \\
\text{val}(\mathcal{M}[\![\textbf{begin } \Delta;\ \Gamma' \textbf{ end}]\!]\langle u, s\rangle) &= \text{val}(\mathcal{M}[\![\Gamma']\!](\mathcal{D}^\dagger[\![\Delta]\!]\langle u', s'\rangle)) \circ \mathcal{T}^\dagger[\![\Delta]\!]\langle u', s'\rangle.
\end{aligned}$$

By Proposition * for declarations, since $X \supseteq \text{free}(\Delta)$,

$$\mathcal{D}^\dagger[\![\Delta]\!]\langle u, s\rangle \approx_{X \cup \text{dec}(\Delta)} \mathcal{D}^\dagger[\![\Delta]\!]\langle u', s'\rangle.$$

Since $X \cup \text{dec}(\Delta) \supseteq \text{free}(\Gamma')$, we can use the induction hypothesis for Γ', getting:

$$\mathcal{M}[\![\Gamma']\!](\mathcal{D}^\dagger[\![\Delta]\!]\langle u, s\rangle) \approx_{X \cup \text{dec}(\Delta)} \mathcal{M}[\![\Gamma']\!](\mathcal{D}^\dagger[\![\Delta]\!]\langle u', s'\rangle).$$

By Proposition (*), $\mathcal{T}[\![\Delta]\!]\langle u, s\rangle = \alpha(\Delta) \circ \text{share}(u)$ and $\mathcal{T}[\![\Delta]\!]\langle u', s'\rangle = \alpha(\Delta) \circ \text{share}(u')$. Hence, for $I \in X$ and $I' \in X \cup \text{dec}(\Delta)$, using the assumption that $\text{share}(u) \Downarrow X = \text{share}(u') \Downarrow X$, we see that

$$(I', I) \in \mathcal{T}^\dagger[\![\Delta]\!]\langle u, s\rangle \quad \Leftrightarrow \quad (I', I) \in \mathcal{T}^\dagger[\![\Delta]\!]\langle u', s'\rangle.$$

The result follows. ∎

Corollary. For all commands Γ and all $\langle u, s\rangle$, $\langle u, s'\rangle$ such that s agrees with s' on $u(\text{free}(\Gamma))$, $\mathcal{S}^\dagger[\![\Gamma]\!]\langle u, s\rangle$ agrees with $\mathcal{S}^\dagger[\![\Gamma]\!]\langle u, s'\rangle$ on $u(\text{free}(\Gamma))$.

These results can be used to prove (for instance) that the effect in any program context of a term is invariant under change of storage allocation primitive, up to sharing equivalence. We omit the details, since this property is not germane to this paper.

It is worth remarking that the *abstract* versions of Invariance and Influence Properties, phrased in terms of $\approx$, may be proved directly, without relying (as we did above) on the concrete versions which mention locations explicitly. If one were to change storage management details, perhaps by altering the semantics of commands so that instead of garbage collecting all inaccessible locations on block exit we merely restrict the store to $\text{dom}(s)$, or even of we omit the restriction, the concrete properties would need to be altered. But the abstract properties still remain true.

Syntactic Substitution.

A *(syntactic) substitution* is a bijection between two finite sets of identifiers. We write $\theta : X \rightarrow X'$ to indicate that the domain and range of the substitution are (respectively) X and X'. We say that such a θ is a substitution on X, and write $\theta : X$ when only the domain of θ is important. Clearly θ^{-1} is also a substitution on X'. We say that θ *affects* I if and only if $\theta(I) \neq I$. If θ is a substitution and Y is a set of identifiers we will abuse notation and write $\theta(Y)$ for $\{\theta(I) \mid I \in Y\}$, and similarly $\theta^{-1}(Y) = \{I \mid \theta(I) \in Y\}$. Clearly $\theta(X \cup Y) = \theta(X) \cup \theta(Y)$ and $\theta(X - Y) = \theta(X) - \theta(Y)$. We use the symbol ι for an identity substitution.

If θ is a substitution on X and E is an expression with $\text{free}(E) \subseteq X$, we will write $[\theta]E$ for the expression obtained by applying θ to all free identifier occurrences of E. Since θ is a bijection there will be no accidental merging of occurrences of previously distinct free identifiers when the substitution is performed. Hence, although we are as usual ignoring the details of expression syntax, we may assume that

$$\text{free}([\theta]E) = \theta(\text{free}(E)).$$

In particular, a substitution applied to an identifier does the obvious: $[\theta]I$ is defined to be $\theta(I)$.

We use a similar notation for substitution on commands: if θ is a substitution on $\text{free}[\![\Gamma]\!]$ we write $[\theta]\Gamma$ for the command resulting when the substitution θ is applied to all free identifier occurrences of Γ. A formal definition of $[\theta]\Gamma$, by structural induction on Γ will follow shortly.

To obtain a satisfactory account of substitution for declarations we need to take into account both bound and free identifier occurrences. It is sometimes necessary to apply one substitution to the free identifier occurrences and another to the bound; two natural special cases are when we wish to change free identifiers but leave declared identifiers fixed, or to change declared identifiers but leave free identifiers fixed. For a fully general definition we will write $(\theta)[\theta']$ for the combination of a substitution θ for the declared identifiers with a substitution θ' for the free identifiers. Thus, $(\theta)[\theta']\Delta$ will be the declaration resulting from the application of this combined substitution to Δ. The special cases mentioned before will then be of the form $(\iota)[\theta']$ and $(\theta)[\iota]$ respectively.

The formal definition is:

Definition. For $\theta : \text{dec}(\Delta)$ and $\theta' : \text{free}(\Delta)$ we define $(\theta)[\theta']\Delta$ by:

$$\begin{aligned}
(\theta)[\theta']\mathbf{null} &= \mathbf{null} \\
(\theta)[\theta'](\mathbf{new}\ I = E) &= (\mathbf{new}\ \theta I = [\theta']E) \\
(\theta)[\theta'](\mathbf{alias}\ I_0 = I_1) &= (\mathbf{alias}\ \theta I_0 = \theta' I_1) \\
(\theta)[\theta'](\Delta_0;\ \Delta_1) &= ((\theta)[\theta']\Delta_0);\ ((\theta)[\theta' + \theta{\downarrow}\text{dec}(\Delta_0)]\Delta_1)
\end{aligned}$$

Clearly, if θ and θ' have overlapping ranges, the combined substitution $(\theta)[\theta']$ on Δ may result in capture of previously free identifier occurrences. Hence the following definition of *safe substitution.*

Definition. The combination $(\theta)[\theta']$ is *safe* for Δ if $\theta' + \theta$ is itself a substitution, or equivalently if

$$\theta(\text{dec}(\Delta)) \cap \theta'(\text{free}(\Delta) - \text{dec}(\Delta)) = \emptyset.$$

The following lemma shows that safe substitutions do not cause capture.

Lemma. If $(\theta)[\theta']$ is safe then

$$\begin{aligned}
\mathrm{free}((\theta)[\theta']\Delta) &= \theta'(\mathrm{free}(\Delta)) \\
\mathrm{dec}((\theta)[\theta']\Delta) &= \theta(\mathrm{dec}(\Delta)) \\
\alpha[\![(\theta)[\theta']\Delta]\!] &= \theta^{-1} \circ \alpha(\Delta) \circ \theta' \\
&= \{(\theta(I'), \theta'(I)) \mid (I', I) \in \alpha[\![\Delta]\!]\}.
\end{aligned}$$

Proof. By structural induction on Δ.

- When $\Delta = \mathbf{null}$ the properties are trivial, since $\mathrm{free}(\mathbf{null}) = \mathrm{dec}(\mathbf{null}) = \alpha(\mathbf{null}) = \emptyset$ and $\theta(\emptyset) = \theta'(\emptyset)$.
- For $\mathbf{new}\ I = E$ we have:

$$(\theta)[\theta'](\mathbf{new}\ I = E) = (\mathbf{new}\ \theta I = [\theta']E),$$

so that clearly

$$\begin{aligned}
\mathrm{free}[\![(\theta)[\theta']\mathbf{new}\ I = E]\!] &= \mathrm{free}([\theta']E), \\
\mathrm{dec}[\![(\theta)[\theta']\mathbf{new}\ I = E]\!] &= \{\theta I\}, \\
\alpha[\![(\theta)[\theta']\mathbf{new}\ I = E]\!] &= \emptyset.
\end{aligned}$$

By assumption, $\mathrm{free}([\theta']E) = \theta'(\mathrm{free}(E))$. The result follows.

- For $\mathbf{alias}\ I_0 = I_1$ we have

$$\begin{aligned}
\mathrm{free}(\mathbf{alias}\ \theta I_0 = \theta' I_1) &= \{\theta' I_1\}, \\
\mathrm{dec}(\mathbf{alias}\ \theta I_0 = \theta' I_1) &= \{\theta I_0\}, \\
\alpha(\mathbf{alias}\ \theta I_0 = \theta' I_1) &= \{(\theta I_0, \theta' I_1)\}
\end{aligned}$$

as required.

- For $\Delta = \Delta_0; \Delta_1$, we have

$$\begin{aligned}
\mathrm{free}(\Delta) &= \mathrm{free}(\Delta_0) \cup (\mathrm{free}(\Delta_1) - \mathrm{dec}(\Delta_0)), \\
\mathrm{dec}(\Delta) &= \mathrm{dec}(\Delta_0) \cup \mathrm{dec}(\Delta_1), \\
\alpha(\Delta) &= (\alpha(\Delta)_0 / \mathrm{dec}(\Delta_1)) \cup (\alpha(\Delta_1) \backslash \mathrm{dec}(\Delta_0)) \cup (\alpha(\Delta_1) \circ \alpha(\Delta_0)).
\end{aligned}$$

Let $\Delta_0' = (\theta)[\theta']\Delta_0$ and $\Delta_1' = (\theta)[\theta' + \theta{\downarrow}\mathrm{dec}(\Delta_0)]\Delta_1$, so that $(\theta)[\theta']\Delta = \Delta_0'; \Delta_1'$. Hence,

$$\begin{aligned}
\mathrm{dec}[\![(\theta)[\theta']\Delta]\!] &= \mathrm{dec}[\![\Delta_0']\!] \cup \mathrm{dec}[\![\Delta_1']\!] \\
&= \theta(\mathrm{dec}(\Delta_0)) \cup \theta(\mathrm{dec}(\Delta_1)),
\end{aligned}$$

by induction hypothesis for Δ_0 and for Δ_1 (using the fact that $(\theta)[\theta']$ safe for $\Delta_0; \Delta_1$ implies $(\theta)[\theta' + \theta{\downarrow}\mathrm{dec}(\Delta_0)]$ safe for Δ_1).

Similarly,

$$\begin{aligned}
\mathrm{free}[\![(\theta)[\theta']\Delta]\!] &= \mathrm{free}[\![\Delta_0']\!] \cup (\mathrm{free}[\![\Delta_1']\!] - \mathrm{dec}[\![\Delta_0']\!]) \\
&= \theta'(\mathrm{free}(\Delta_0)) \cup [(\theta' + \theta{\downarrow}\mathrm{dec}(\Delta_0))(\mathrm{free}(\Delta_1)) - \theta(\mathrm{dec}(\Delta_0))] \\
&= \theta'(\mathrm{free}(\Delta_0)) \cup \theta'(\mathrm{free}(\Delta_1) - \mathrm{dec}(\Delta_0)) \\
&= \theta'(\mathrm{free}(\Gamma)
\end{aligned}$$

as required.

Finally,

$$\alpha((\theta)[\theta']\Delta) \;=\; (\alpha(\Delta_0')/\mathrm{dec}(\Delta_1')) \cup (\alpha(\Delta_1')\backslash\mathrm{dec}(\Delta_0')) \cup (\alpha(\Delta_1') \circ \alpha(\Delta_0')).$$

By the induction hypotheses,

$$\begin{aligned}\alpha(\Delta_0')/\mathrm{dec}(\Delta_1') &= \{(\theta I', \theta' I) \mid (I', I) \in \alpha(\Delta_0)\}/\theta(\mathrm{dec}(\Delta_1)) \\ &= \{(\theta I', \theta' I) \mid (I', I) \in \alpha(\Delta_0) \;\;\&\;\; I' \notin \mathrm{dec}(\Delta_1)\} \\ &= \theta^{-1} \circ (\alpha(\Delta_0)/\mathrm{dec}(\Delta_1)) \circ \theta'.\end{aligned}$$

Similarly,

$$\alpha(\Delta_1')\backslash\mathrm{dec}(\Delta_0') \;=\; \theta^{-1} \circ (\alpha(\Delta_1)\backslash\mathrm{dec}(\Delta_0)) \circ \theta'.$$

And

$$\begin{aligned}\alpha(\Delta_1') \circ \alpha(\Delta_0') &= (\theta^{-1} \circ \alpha(\Delta_1) \circ (\theta' + \theta{\downarrow}\mathrm{dec}(\Delta_0))) \circ (\theta^{-1} \circ \alpha(\Delta_0) \circ \theta') \\ &= \theta^{-1} \circ \alpha(\Delta_1) \circ \alpha(\Delta_0) \circ \theta'\end{aligned}$$

because $(\theta' + \theta{\downarrow}\mathrm{dec}(\Delta_0)) \circ \theta{\downarrow}\mathrm{dec}(\Delta_0)$ is the identity on $\mathrm{dec}(\Delta_0)$.

The result follows easily. ∎

Now we are ready to establish the proper semantic behavior of substitution.

If $\theta : X \to X'$ is a substitution and u is an environment defined on X, then $u \circ \theta^{-1}$ is an environment defined on X', with the obvious characteristic property that $u \circ \theta^{-1}$ treats the image of I under θ exactly as u treats I. Hence, any substitution $\theta : X \to X'$ determines a function $C_\theta : C_X \to C_{X'}$ by

$$C_\theta\langle u, s\rangle \;=\; \langle u \circ \theta^{-1}, s\rangle.$$

And, for identifiers I it is trivial that for all $\langle u, s\rangle$:

$$\begin{aligned}&\mathcal{E}[\![[\theta]I]\!]\langle u \circ \theta^{-1}, s\rangle \;= \mathcal{E}[\![I]\!]\langle u, s\rangle, \\ &\text{i.e.,}\quad \mathcal{E}[\![[\theta]I]\!] \circ C_\theta \;=\; \mathcal{E}[\![I]\!].\end{aligned}$$

We make the obvious assumption that this property extends to all expressions:

Assumption 2. For all E and all θ defined on $\mathrm{free}[\![E]\!]$,

$$\mathcal{E}[\![[\theta]E]\!] \circ C_\theta \;=\; \mathcal{E}[\![E]\!]. \;\;∎$$

The proper semantic effect of syntactic substitution on declarations is expressed as follows.

Proposition S1. If $(\theta)[\theta']$ is safe for Δ, then

$$\mathcal{D}[\![(\theta)[\theta']\Delta]\!] \circ C_{\theta'} = C_\theta \circ \mathcal{D}[\![\Delta]\!].$$

Proof. By structural induction on Δ.

Case (a). For $\Delta = \mathbf{null}$ we merely require that for all θ and θ',

$$\mathcal{D}[\![\mathbf{null}]\!] \circ C_{\theta'} \;=\; C_\theta \circ \mathcal{D}[\![\mathbf{null}]\!].$$

This is trivially true, since $\mathcal{D}[\![\mathbf{null}]\!]\langle u, s\rangle = \langle \emptyset, \emptyset\rangle$.

Case (b). For Δ of the form **new** $I = E$, let $\langle u, s\rangle \in C_{\text{free}(E)}$. We have:

$$\begin{aligned}
(\mathcal{D}[\![(\theta)[\theta']\Delta]\!] \circ C_{\theta'})\langle u, s\rangle &= \mathcal{D}[\![\textbf{new } \theta(I) = [\theta']E]\!]\langle u \circ \theta'^{-1}, s\rangle \\
&= \langle [\theta(I) \mapsto l], [l \mapsto \mathcal{E}[\![[\theta']E]\!]\langle u \circ \theta'^{-1}, s\rangle]\rangle \\
&\qquad\qquad \text{where } l = \text{new}(\text{dom}(s)) \\
&= \langle [\theta(I) \mapsto l], [l \mapsto \mathcal{E}[\![E]\!]\langle u, s\rangle]\rangle \\
&\qquad\qquad \text{by Assumption} \\
&= \langle [I \mapsto l] \circ \theta^{-1}, [l \mapsto \mathcal{E}[\![E]\!]\langle u, s\rangle]\rangle \\
&= C_\theta\langle [I \mapsto l], [l \mapsto \mathcal{E}[\![E]\!]\langle u, s\rangle]\rangle \\
&= C_\theta(\mathcal{D}[\![\textbf{new } I = E]\!]\langle u, s\rangle) \qquad \text{as required.}
\end{aligned}$$

That completes the analysis for a **new** declaration.

Case (c). For Δ of the form **alias** $I_0 = I_1$, we have:

$$\begin{aligned}
\mathcal{D}[\![(\theta)[\theta']\textbf{alias } I_0 = I_1]\!]\langle u \circ \theta'^{-1}, s\rangle &= \mathcal{D}[\![\textbf{alias } \theta(I_0) = \theta'(I_1)]\!]\langle u \circ \theta'^{-1}, s\rangle \\
&= \langle [\theta(I_0) \mapsto (u \circ \theta'^{-1})(\theta'(I_1))], s{\downarrow}(u \circ \theta'^{-1}(\theta'(I_1)))\rangle \\
&= \langle [\theta(I_0) \mapsto u(I_1)], s{\downarrow}u(I_1)\rangle \\
&= \langle [I_0 \mapsto u(I_1)] \circ \theta^{-1}, s{\downarrow}u(I_1)\rangle \\
&= C_\theta\langle [I_0 \mapsto u(I_1), s{\downarrow}u(I_1)\rangle \\
&= C_\theta(\mathcal{D}[\![\textbf{alias } I_0 = I_1]\!]\langle u, s\rangle)
\end{aligned}$$

as required.

Case (d). For Δ of form $\Delta_0;\ \Delta_1$, let

$$\begin{aligned}
\langle u_0, s_0\rangle &= \mathcal{D}[\![\Delta_0]\!]\langle u, s\rangle, \\
\langle u_1, s_1\rangle &= \mathcal{D}[\![\Delta_1]\!]\langle u + u_0, s + s_0\rangle, \\
\text{so that} \quad \mathcal{D}[\![\Delta_0;\ \Delta_1]\!]\langle u, s\rangle &= \langle u_0 + u_1, s_0 + s_1\rangle.
\end{aligned}$$

Let $\theta'' = \theta' + \theta{\downarrow}\text{dec}(\Delta_0)$, and

$$\begin{aligned}
\langle u_0', s_0'\rangle &= \mathcal{D}[\![(\theta)[\theta']\Delta_0]\!]\langle u \circ \theta'^{-1}, s\rangle, \\
\langle u_1', s_1'\rangle &= \mathcal{D}[\![(\theta)[\theta'']\Delta_1]\!]\langle u \circ \theta'^{-1} + u_0', s + s_0'\rangle,
\end{aligned}$$

so that

$$\begin{aligned}
\mathcal{D}[\![(\theta)[\theta'](\Delta_0;\ \Delta_1)]\!]\langle u \circ \theta'^{-1}, s\rangle &= \mathcal{D}[\![(\theta)[\theta']\Delta_0;\ (\theta)[\theta'']\Delta_1]\!]\langle u \circ \theta'^{-1}, s\rangle \\
&= \langle u_0' + u_1', s_0' + s_1'\rangle
\end{aligned}$$

We want to prove that $s_0 + s_1 = s_0' + s_1'$ and that $(u_0 + u_1) \circ \theta^{-1} = u_0' + u_1'$.

By induction hypothesis for Δ_0,

$$\langle u_0', s_0{\prime}\rangle = \langle u_0 \circ \theta^{-1}, s_0\rangle.$$

Thus, $u_0' = u_0 \circ \theta^{-1}$ and $s_0' = s_0$.

The induction hypothesis for Δ_1 is not immediately applicable, since in general we do not have the identity $u \circ \theta'^{-1} + u_0 \circ \theta^{-1} = (u + u_0) \circ \theta''^{-1}$. However, these two environments

agree on the free identifiers of $(\theta)[\theta'']\Delta_1$, i.e. on $\theta'(\text{free}(\Delta_1) - \text{dec}(\Delta_0)) \cup \theta(\text{free}(\Delta_1) \cap \text{dec}(\Delta_0))$. That is enough to guarantee that

$$\mathcal{D}[\![(\theta)[\theta'']\Delta_1]\!]\langle u \circ \theta'^{-1} + u_0^*, s + s_0\rangle \;=\; \mathcal{D}[\![(\theta)[\theta'']\Delta_1]\!]\langle (u + u_0) \circ (\theta'')^{-1}, s + s_0\rangle,$$

by the Agreement Property for declarations. Now we have a form to which we may apply the inductive hypothesis, yielding

$$\langle u_1', s_1'\rangle \;=\; \langle u_1 \circ \theta^{-1}, s_1\rangle.$$

The result follows, since $(u_0 + u_1) \circ \theta^{-1} = (u_0 \circ \theta^{-1} + u_1 \circ \theta^{-1})$. That completes the proof. ∎

A corresponding property for $\mathcal{D}^\dagger$ is easily derivable: in general, when $(\theta)[\theta']$ is safe,

$$\begin{aligned}
\mathcal{R}^\dagger[\![(\theta)[\theta']\Delta]\!]\langle u \circ \theta'^{-1}, s\rangle &\;=\; u \circ \theta'^{-1} + (\mathcal{R}[\![\Delta]\!]\langle u, s\rangle) \circ \theta^{-1},\\
\mathcal{S}^\dagger[\![(\theta)[\theta']\Delta]\!]\langle u \circ \theta'^{-1}, s\rangle &\;=\; \mathcal{S}^\dagger[\![\Delta]\!]\langle u, s\rangle.
\end{aligned}$$

In the special case when $\text{dec}(\Delta)$ is disjoint from $\text{dom}(u)$ we obtain the following simple property:

Lemma. If $(\theta)[\theta']$is safe and $\text{dec}(\Delta) \cap \text{dom}(u) = \emptyset$ then

$$\mathcal{D}^\dagger[\![(\theta)[\theta']\Delta]\!] \circ C_{\theta'} \;=\; C_{\theta' + \theta\downarrow\text{dec}(\Delta)} \circ \mathcal{D}^\dagger[\![\Delta]\!].$$

Proof. By the previous result, if $\mathcal{D}[\![\Delta]\!]\langle u, s\rangle = \langle u_0, s_0\rangle$ then

$$\mathcal{D}[\![(\theta)[\theta']\Delta]\!]\langle u \circ \theta'^{-1}, s\rangle \;=\; \langle u_0 \circ \theta^{-1}, s_0\rangle.$$

By definition,

$$\mathcal{D}^\dagger[\![(\theta)[\theta']\Delta]\!]\langle u \circ \theta'^{-1}, s\rangle \;=\; \langle u \circ \theta'^{-1} + u_0 \circ \theta^{-1}, s + s_0\rangle.$$

We must show that $(u + u_0) \circ (\theta' + \theta\downarrow\text{dec}(\Delta))^{-1} \;=\; u \circ \theta'^{-1} + u_0 \circ \theta^{-1}$. Both of these environments have domain $\theta(\text{dec}(\Delta)) \cup \theta'(\text{dom}(u) - \text{dec}(\Delta)) = \theta(\text{dec}(\Delta)) \cup \theta'(\text{dom}(u))$ by assumption.

Now $\text{dom}(u_0) = \text{dec}(\Delta)$, so that $u_0 \circ \theta^{-1}$ is defined only on $\text{rge}(\theta\downarrow\text{dec}(\Delta))$. Clearly, if $I \in \text{dec}(\Delta)$ we get

$$\begin{aligned}
((u + u_0) \circ (\theta' + \theta\downarrow\text{dec}(\Delta))^{-1})(\theta I) &= u_0(I)\\
&= (u \circ \theta'^{-1} + u_0 \circ \theta^{-1})(\theta I).
\end{aligned}$$

And for $I \in \text{dom}(u)$ we have

$$\begin{aligned}
((u + u_0) \circ (\theta' + \theta\downarrow\text{dec}(\Delta))^{-1})(\theta' I) &= (u + u_0)(I)\\
&= u(I)\\
&= (u \circ \theta'^{-1} + u_0 \circ \theta^{-1})(\theta' I).
\end{aligned}$$

As a counterexample to show the necessity of the disjointness constraint on Δ, choose $u = [x \mapsto l]$, $u_0 = [x \mapsto l_0]$ (coming perhaps from $\Delta = \textbf{new } x = 0$), $\theta x = y$, $\theta' x = x$. Then $(u + u_0) \circ (\theta' + \theta)^{-1} \;=\; [y \mapsto l_0]$, whereas $u \circ \theta'^{-1} + u_0 \circ \theta^{-1} \;=\; [x \mapsto l, y \mapsto l_0]$.

For commands we specify free substitution as follows: The definition is straightforward, except for the renaming of bound identifiers in a block in order to avoid capture.

$$\begin{aligned}
[\theta]\mathbf{skip} &= \mathbf{skip} \\
[\theta](I{:=}E) &= [\theta]I{:=}[\theta]E \\
[\theta](\Gamma_0;\ \Gamma_1) &= [\theta]\Gamma_0;\ [\theta]\Gamma_1 \\
[\theta]\mathbf{begin}\ \Delta;\ \Gamma\ \mathbf{end} &= \mathbf{begin}\ (\eta)[\theta]\Delta;\ [\theta+\eta]\Gamma\ \mathbf{end},
\end{aligned}$$

where η is a substitution on $\mathrm{dec}[\![\Delta]\!]$ chosen to make $(\eta)[\theta]$ a safe substitution. There are of course many such η, but we will show shortly that the choice is semantically irrelevant. However, it would perhaps be worth mentioning that a completely unambiguous definition of substitution can be obtained by choosing a particular renaming function *newsubst* that, given a substitution θ and a finite set of identifiers X returns a substitution on X consistent with θ on all identifiers in $\mathrm{dom}(\theta) \cap X$. Then we may always choose η to be $\mathit{newsubst}(\theta, \mathrm{dec}[\![\Delta]\!])$. This use of an auxiliary function satisfying a newness condition is of course reminiscent of what was done earlier in the location semantics with the *new* function.

To express the proper behavior of substitution on commands we state:

Proposition S2. For all Γ and all substitutions θ defined on $\mathrm{free}(\Gamma)$,

$$\mathcal{M}[\![[\theta]\Gamma]\!] \circ C_\theta = C_\theta \circ \mathcal{M}[\![\Gamma]\!].$$

Proof. By structural induction on Γ.

Case (a). For $\Gamma = \mathbf{skip}$ the property is trivial.

Case (b). For Γ of the form $I{:=}E$, we have

$$\begin{aligned}
\mathcal{M}[\![[\theta](I{:=}E)]\!]\langle u \circ \theta^{-1}, s\rangle &= \mathcal{M}[\![\theta(I){:=}[\theta]E]\!]\langle u \circ \theta^{-1}, s\rangle \\
&= \langle u \circ \theta^{-1}, s + [u \circ \theta^{-1}(\theta(I)) \mapsto \mathcal{E}[\![[\theta]E]\!]\langle u \circ \theta^{-1}, s\rangle]\rangle \\
&= \langle u \circ \theta^{-1}, s + [u(I) \mapsto \mathcal{E}[\![E]\!]\langle u, s\rangle]\rangle \\
&= C_\theta(\mathcal{M}[\![I{:=}E]\!]\langle u, s\rangle)
\end{aligned}$$

as required.

Case (c). For Γ of the form $\Gamma_0;\ \Gamma_1$ we have:

$$\begin{aligned}
\mathcal{M}[\![[\theta]\Gamma]\!] \circ C_\theta &= \mathcal{M}[\![[\theta]\Gamma_0;\ [\theta]\Gamma_1]\!] \circ C_\theta \\
&= \mathcal{M}[\![[\theta]\Gamma_1]\!] \circ \mathcal{M}[\![[\theta]\Gamma_0]\!] \circ C_\theta \\
&= \mathcal{M}[\![[\theta]\Gamma_1]\!] \circ C_\theta \circ \mathcal{M}[\![\Gamma_0]\!] && \text{by hypothesis for } \Gamma_0 \\
&= C_\theta \circ \mathcal{M}[\![\Gamma_1]\!] \circ \mathcal{M}[\![\Gamma_0]\!] && \text{by hypothesis for } \Gamma_1 \\
&= C_\theta \circ \mathcal{M}[\![\Gamma_0;\ \Gamma_1]\!] && \text{as required.}
\end{aligned}$$

Case (d). For Γ of form $\mathbf{begin}\ \Delta_0;\ \Gamma_0\ \mathbf{end}$ we have:

$$\mathcal{M}[\![[\theta]\mathbf{begin}\ \Delta_0;\ \Gamma_0\ \mathbf{end}]\!]\langle u \circ \theta^{-1}, s\rangle = \mathcal{M}[\![\mathbf{begin}\ (\eta)[\theta]\Delta_0;\ [\theta+\eta]\Gamma\ \mathbf{end}]\!]\langle u \circ \theta^{-1}, s\rangle.$$

Let $\langle u_0, s_0\rangle = \mathcal{D}^\dagger[\![\Delta_0]\!]\langle u, s\rangle$ and $\langle u_0, s_1\rangle = \mathcal{M}[\![\Gamma_0]\!]\langle u_0, s_0\rangle$. By definition, $\mathcal{M}[\![\Gamma]\!]\langle u, s\rangle = \langle u, s_1{\downarrow}\mathrm{rge}(u)\rangle$. By induction hypotheses we have:

$$\begin{aligned}
\mathcal{D}^\dagger[\![(\eta)[\theta]\Delta_0]\!]\langle u \circ \theta^{-1}, s\rangle &= C_{\theta+\eta}(\mathcal{D}^\dagger[\![\Delta_0]\!]\langle u, s\rangle) \\
&= \langle u_0 \circ (\theta+\eta)^{-1}, s_0\rangle \\
\mathcal{M}[\![[\theta+\eta]\Gamma_0]\!](C_{\theta+\eta}(\mathcal{D}^\dagger[\![\Delta_0]\!]\langle u, s\rangle)) &= C_{\theta+\eta}(\mathcal{M}[\![\Gamma_0]\!](\mathcal{D}[\![\Delta_0]\!]\langle u, s\rangle)) \\
&= \langle u_0 \circ (\theta+\eta)^{-1}, s_1\rangle
\end{aligned}$$

Hence,

$$\mathcal{M}[\![[\theta]\Gamma]\!]\langle u \circ \theta^{-1}, s\rangle \;=\; \langle u \circ \theta^{-1}, s_1{\downarrow}\mathrm{rge}(u \circ \theta^{-1})\rangle.$$

But since θ is a substitution on $\mathrm{dom}(u)$, $\mathrm{rge}(u) = \mathrm{rge}(u \circ \theta^{-1})$. The result follows easily. ∎

Corollary. If $\eta : \mathrm{dec}(\Delta)$ has range disjoint from $\mathrm{dom}(u)$ then

$$\mathcal{M}[\![\mathbf{begin}\ \Delta;\ \Gamma\ \mathbf{end}]\!]\langle u, s\rangle = \mathcal{M}[\![\mathbf{begin}\ (\eta)[\iota]\Delta;\ [\iota+\eta]\Gamma\ \mathbf{end}]\!]\langle u, s\rangle.$$

Proof. By previous results. ∎

Corollary. If $\eta : \mathrm{dec}(\Delta)$ makes $(\eta)[\iota]$ safe, then

$$\mathcal{M}[\![\mathbf{begin}\ \Delta;\ \Gamma\ \mathbf{end}]\!]\langle u, s\rangle = \mathcal{M}[\![\mathbf{begin}\ (\eta)[\iota]\Delta;\ [\iota+\eta]\Gamma\ \mathbf{end}]\!]\langle u, s\rangle.$$

Proof.

$$\begin{aligned}
\mathcal{M}[\![\mathbf{begin}\ (\eta)\iota]\Delta;\ [\iota+\eta]\Gamma\ \mathbf{end}]\!]\langle u, s\rangle &= \mathcal{M}[\![[\iota]\mathbf{begin}\ \Delta;\ \Gamma\ \mathbf{end}]\!]\langle u, s\rangle \\
&= \mathcal{M}[\![[\iota]\mathbf{begin}\ \Delta;\ \Gamma\ \mathbf{end}]\!]\langle u \circ \iota^{-1}, s\rangle \\
&= \mathcal{M}[\![\mathbf{begin}\ \Delta;\ \Gamma\ \mathbf{end}]\!]\langle u \circ \iota^{-1}, s\rangle \\
&= \mathcal{M}[\![\mathbf{begin}\ \Delta;\ \Gamma\ \mathbf{end}]\!]\langle u, s\rangle.
\end{aligned}$$

∎

To show the necessity of these constraints:

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 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 $z{:=}z+1$. ∎

4. 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 $y{:=}y+1$. ∎

5. Finally, in the block

begin null; $x{:=}1$ **end**

the identifier x is not declared. If we rename x to y we get

begin null; $y{:=}1$ **end**,

which is not semantically equivalent to the original block. ∎

Abstract Semantics.

The standard semantics was based on *configurations*; we have seen that the semantic effect of all terms in our language is determined uniquely up to sharing equivalence. It is therefore possible to factor out the standard semantics by means of this equivalence and obtain a semantics based directly on equivalence classes of configurations, in which sharing relations and valuations are the primary objects of construction. To distinguish between this "abstract" semantics and the standard one, we will use $\overline{\mathcal{E}}, \overline{\mathcal{D}}, \overline{\mathcal{M}}$, and $\overline{\mathcal{P}}$ for the abstract semantic functions. First some notation.

We let ρ range over the set R of *sharing relations*, which are equivalence relations on a finite set of identifiers. The domain of ρ is defined to be the set of identifiers involved in ρ, which may be described simply as $\mathrm{dom}(\rho) = \{I \mid (I, I) \in \rho\}$. We let σ range over the set Σ of *valuations*, which are finite partial functions from **Ide** to V. A pair $\langle \rho, \sigma \rangle$ is *consistent* iff

$$\forall (I_0, I_1) \in \rho.\ [\sigma(I_0) = \sigma(I_1)].$$

Equally well, when σ is consistent with ρ it determines a total map from the equivalence classes of ρ to V.

We refer to consistent pairs $\langle \rho, \sigma \rangle$ like this as *frames*, and we let f range over the set F of frames. Where necessary, F_X denotes the set of frames defined on (a superset of) X, R_X denotes the set of sharing relations whose domain includes X, and Σ_X denotes the set of valuations whose domain includes X. Frames are essentially abstract configurations, representatives of equivalence classes of configurations.

We now introduce the notation

$$\overline{\langle u, s \rangle} \ = \ \langle \mathrm{share}(u), \mathrm{val}(u, s) \rangle.$$

It is clear that every configuration $\langle u, s \rangle$ determines a consistent pair $\langle \rho, \sigma \rangle = \overline{\langle u, s \rangle}$, with $\mathrm{dom}(\rho) = \mathrm{dom}(\sigma) = \mathrm{dom}(u)$. Clearly, a configuration in C_X determines a frame in F_X. Moreover, every frame $\langle \rho, \sigma \rangle$ in F_X is obtainable as $\overline{\langle u, s \rangle}$ for a suitably chosen configuration $\langle u, s \rangle$ in C_X.

The abstract semantic functions are:

$$\begin{aligned}
&\overline{\mathcal{E}}[\![E]\!] : F_{\mathrm{free}(E)} \to V \\
&\overline{\mathcal{D}}[\![\Delta]\!] : F_{\mathrm{free}(\Delta)} \to F_{\mathrm{dec}(\Delta)} \\
&\overline{\mathcal{M}}[\![\Gamma]\!] : F_{\mathrm{free}(\Gamma)} \to F_{\mathrm{free}(\Gamma)} \\
&\overline{\mathcal{P}} : \mathbf{Prog} \to V.
\end{aligned}$$

They may be defined implicitly with respect to the standard semantics, as follows:

$$\overline{\mathcal{E}}[\![E]\!]\overline{\langle u, s \rangle} \ = \ \mathcal{E}[\![E]\!]\langle u, s \rangle$$

$$\overline{\mathcal{D}}[\![\Delta]\!]\overline{\langle u, s \rangle} \ = \ \overline{\mathcal{D}[\![\Delta]\!]\langle u, s \rangle}$$

$$\overline{\mathcal{M}}[\![\Gamma]\!]\overline{\langle u, s \rangle} \ = \ \overline{\mathcal{M}[\![\Gamma]\!]\langle u, s \rangle}$$

$$\overline{\mathcal{P}}[\![\mathbf{begin}\ \Delta;\ \Gamma;\ \mathbf{result}\ E\ \mathbf{end}]\!] \ = \ \overline{\mathcal{E}}[\![E]\!](\overline{\mathcal{M}}[\![\Gamma]\!](\overline{\mathcal{D}}[\![\Delta]\!]\langle \emptyset, \emptyset \rangle)).$$

These definitions are unambiguous, and suffice to define the semantic functions on their entire domains as given above. In fact, we will show soon that $\overline{\mathcal{P}} = \mathcal{P}$.

As with the standard semantics, we introduce auxiliary functions describing the separate effects of a declaration on the sharing relationship and on the values of identifiers. However, in this abstract setting we can slightly simplify the types of these auxiliaries, since it is easy to see that the effect of a declaration on the sharing relation depends only on the sharing relation, and the effect of a declaration on the valuation depends only on the values of identifiers: thus, we put

$$\overline{\mathcal{R}}[\![\Delta]\!] : R_{\mathrm{free}(\Delta)} \to R_{\mathrm{dec}(\Delta)},$$
$$\overline{\mathcal{S}}[\![\Delta]\!] : \Sigma_{\mathrm{free}(\Delta)} \to \Sigma_{\mathrm{dec}(\Delta)},$$

with the definitions being:

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

We also define $\overline{\mathcal{D}^\dagger}$ and its two component functions $\overline{\mathcal{R}^\dagger}, \overline{\mathcal{S}^\dagger}$ by

$$\begin{aligned}
\overline{\mathcal{D}^\dagger}[\![\Delta]\!]\overline{\langle u,s\rangle} &= \overline{\mathcal{D}^\dagger[\![\Delta]\!]\langle u,s\rangle} \\
\langle\overline{\mathcal{R}^\dagger}[\![\Delta]\!]\rho, \overline{\mathcal{S}^\dagger}[\![\Delta]\!]\sigma\rangle &= \overline{\mathcal{D}^\dagger}[\![\Delta]\!]\langle\rho,\sigma\rangle.
\end{aligned}$$

Given a pair of sharing relations ρ and ρ', and a relation $\alpha \subseteq \mathrm{dom}(\rho') \times \mathrm{dom}(\rho)$, we will define

$$\begin{aligned}
\rho +_\alpha \rho' \;&= \rho|\mathrm{dom}(\rho') \cup \rho' \\
&\quad \cup \{(I,I'),(I',I) \mid I \notin \mathrm{dom}(\rho') \;\&\; (I',I) \in \rho' \circ \alpha \circ \rho\}.
\end{aligned}$$

Then we have the identity:

$$\mathrm{share}(\mathcal{R}^\dagger[\![\Delta]\!]\langle u,s\rangle) \;=\; \mathrm{share}(u) +_{\alpha(\Delta)} \mathrm{share}(\mathcal{R}[\![\Delta]\!]\langle u,s\rangle).$$

Hence, $\overline{\mathcal{R}^\dagger}[\![\Delta]\!]\rho \;=\; \rho +_{\alpha(\Delta)} \overline{\mathcal{R}}[\![\Delta]\!]\rho$. In the special case where $\alpha(\Delta) = \emptyset$ and $\mathrm{dec}(\Delta) \cap \mathrm{dom}(\rho) = \emptyset$ we simply get $\overline{\mathcal{R}^\dagger}[\![\Delta]\!]\rho = \rho \cup \overline{\mathcal{R}}[\![\Delta]\!]\rho$.

The abstract version of $\mathcal{T}$ is simply

$$\overline{\mathcal{T}}[\![\Delta]\!]\rho \;=\; (\overline{\mathcal{R}}[\![\Delta]\!]\rho) \circ \alpha\Delta \circ \rho.$$

Similarly,

$$\begin{aligned}
\overline{\mathcal{T}^\dagger}[\![\Delta]\!]\rho \;&= (\overline{\mathcal{R}^\dagger}[\![\Delta]\!]\rho) \circ \alpha\Delta \circ \rho \\
&= \rho|\mathrm{dec}[\![\Delta]\!] \cup \overline{\mathcal{T}}[\![\Delta]\!]\rho.
\end{aligned}$$

Next we give a set of equations defining the abstract semantics of a term as a function of the abstract semantics of its immediate syntactic subterms. In other words, a denotational semantics. All of these equations are derivable directly from the semantic definitions given above.

$$\overline{\mathcal{E}}[\![I]\!]\langle\rho,\sigma\rangle \;=\; \sigma(I)$$

$$\begin{aligned}
\overline{\mathcal{D}^\dagger}[\![\mathbf{null}]\!]\langle\rho,\sigma\rangle &= \langle\rho,\sigma\rangle \\
\overline{\mathcal{D}^\dagger}[\![\mathbf{new}\ I = E]\!]\langle\rho,\sigma\rangle &= \langle\rho + [I], \sigma + [I \mapsto \overline{\mathcal{E}}[\![E]\!]\langle\rho,\sigma\rangle]\rangle \\
\overline{\mathcal{D}^\dagger}[\![\mathbf{alias}\ I_0 = I_1]\!]\langle\rho,\sigma\rangle &= \langle\rho + [I_0 \mapsto I_1], \sigma + [I_0 \mapsto \sigma(I_1)]\rangle \\
\overline{\mathcal{D}^\dagger}[\![\Delta_0;\ \Delta_1]\!]\langle\rho,\sigma\rangle &= \overline{\mathcal{D}^\dagger}[\![\Delta_1]\!](\overline{\mathcal{D}^\dagger}[\![\Delta_0]\!]\langle\rho,\sigma\rangle))
\end{aligned}$$

where we use the notation $\rho + [I]$ for the sharing relation obtained from ρ by removing I from its sharing class and inserting it in a trivial class by itself, and $\rho + [I_0 \mapsto I_1]$ for the sharing relation obtained from ρ by removing I_0 from its sharing class and inserting it into the sharing class of I_1:

$$\begin{aligned} \rho + [I] &= \rho +_{\emptyset} \{(I, I)\} = \{(I_1, I_2) \in \rho \mid I_1 \neq I \ \& \ I_2 \neq I\} \cup \{(I, I)\} \\ \rho + [I_0 \mapsto I_1] &= \rho +_{[I_0 \mapsto I_1]} \{(I_0, I_0)\} = (\rho + [I_0]) \cup \{(I_0, I), (I, I_0) \mid (I_1, I) \in \rho\}. \end{aligned}$$

The notation $\sigma + [I \mapsto v]$ is the usual function overwriting.

We also inherit at this abstract level the analogues of the (abstract versions) of the semantic properties proven earlier for the standard semantics (*modulo* the abstract versions of the assumptions we made about the semantics of expressions).

We first define an appropriate notion of *agreement* on frames. Two frames agree on a finite set of identifiers X iff they describe the same sharing relation on X and give the same values to all identifiers in X:

$$\langle \rho, \sigma \rangle \equiv_X \langle \rho', \sigma' \rangle \quad \Leftrightarrow \quad (\rho \Downarrow X = \rho' \Downarrow X) \ \& \ (\sigma \downarrow X = \sigma' \downarrow X).$$

Clearly, $\langle u, s \rangle \approx_X \langle u', s' \rangle$ iff $\overline{\langle u, s \rangle} \equiv_X \overline{\langle u', s' \rangle}$. Two frames are identical if they have the same domain and agree on this domain.

Assumption. For all E and all $\langle \rho, \sigma \rangle$ and $\langle \rho', \sigma' \rangle$ in $F_{\mathrm{free}(E)}$

$$\langle \rho, \sigma \rangle \equiv_X \langle \rho', \sigma' \rangle \quad \Rightarrow \quad \overline{\mathcal{E}}[\![E]\!]\langle \rho, \sigma \rangle = \overline{\mathcal{E}}[\![E]\!]\langle \rho', \sigma' \rangle. \ \blacksquare$$

Theorem 1. For all Δ and all frames $\langle \rho, \sigma \rangle$ defined on $\mathrm{free}(\Delta)$,

$$\overline{\mathcal{D}^{\dagger}}[\![\Delta]\!]\langle \rho, \sigma \rangle \equiv \langle \rho, \sigma \rangle \quad \text{on } \mathrm{dom}(\rho) - \mathrm{dec}(\Delta).$$

Theorem 2. For all Δ, if f and f' agree on $\mathrm{free}[\![\Delta]\!]$ then

$$\begin{aligned} \overline{\mathcal{D}}[\![\Delta]\!]f &\equiv_{\mathrm{dec}(\Delta)} \overline{\mathcal{D}}[\![\Delta]\!]f' \\ \overline{\mathcal{D}^{\dagger}}[\![\Delta]\!]f &\equiv_{\mathrm{ids}(\Delta)} \overline{\mathcal{D}^{\dagger}}[\![\Delta]\!]f'. \ \blacksquare \end{aligned}$$

Commands. The abstract semantics of commands is characterized by the following equations:

$$\begin{aligned} \overline{\mathcal{M}}[\![\mathbf{skip}]\!]\langle \rho, \sigma \rangle &= \langle \rho, \sigma \rangle \\ \overline{\mathcal{M}}[\![I{:=}E]\!]\langle \rho, \sigma \rangle &= \langle \rho, \sigma + [\rho(I) \mapsto \mathcal{E}[\![E]\!]\sigma] \rangle \\ \overline{\mathcal{M}}[\![\Gamma_0;\ \Gamma_1]\!]\langle \rho, \sigma \rangle &= \overline{\mathcal{M}}[\![\Gamma_1]\!](\overline{\mathcal{M}}[\![\Gamma_0]\!]\langle \rho, \sigma \rangle) \\ \overline{\mathcal{M}}[\![\mathbf{begin}\ \Delta;\ \Gamma\ \mathbf{end}]\!]\langle \rho, \sigma \rangle &= \langle \rho, \sigma + \sigma' \circ \tau \rangle, \\ \text{where} \quad \sigma' &= \overline{\mathcal{S}^{\dagger}}[\![\Gamma]\!](\overline{\mathcal{D}^{\dagger}}[\![\Delta]\!]\langle \rho, \sigma \rangle) \\ \text{and} \quad \tau &= \overline{\mathcal{T}^{\dagger}}[\![\Delta]\!]\rho. \end{aligned}$$

When $\mathrm{dec}(\Delta)$ is disjoint from $\mathrm{dom}(\rho)$, the equation for a block can be greatly simplified to:

$$\overline{\mathcal{M}}[\![\Gamma]\!](\overline{\mathcal{D}^{\dagger}}[\![\Delta]\!]\langle \rho, \sigma \rangle)\downarrow\rho,$$

where we use the notation $\langle \rho', \sigma' \rangle \downarrow \rho$ for $\langle \rho, \sigma' \downarrow \mathrm{dom}(\rho) \rangle$. In the case where $\mathrm{dec}[\![\Delta]\!] \cap \mathrm{dom}(\rho)$ is empty, $\overline{\mathcal{T}^{\dagger}}[\![\Delta]\!]\rho$ simply behaves like an identity function on $\mathrm{dom}(\rho)$, in that for all $I \in$

$\mathrm{dom}(\rho)$, $(I, I) \in \mathcal{T}[\![\Delta]\!]\rho$; thus, composition with this function is equivalent to restriction to $\mathrm{dom}(\rho)$.

An alternative treatment of blocks is suggested if we take advantage of the result that the standard semantics of blocks is invariant under safe substitution, so that:

$$\overline{\mathcal{M}}[\![\mathbf{begin}\ \Delta;\ \Gamma\ \mathbf{end}]\!]\langle\rho,\sigma\rangle \ = \ \overline{\mathcal{M}}[\![\mathbf{begin}\ (\eta)[\iota]\Delta;\ [\iota+\eta]\Gamma\ \mathbf{end}]\!]\langle\rho,\sigma\rangle,$$

whenever η is a substitution on $\mathrm{dec}[\![\Delta]\!]$ with range disjoint from $\mathrm{dom}(\rho)$. This guarantees that the first equation will be applicable to the substituted block, so that we may define

$$\overline{\mathcal{M}}[\![\mathbf{begin}\ \Delta;\ \Gamma\ \mathbf{end}]\!]\langle\rho,\sigma\rangle \ = \ \overline{\mathcal{M}}[\![[\iota+\eta]\Gamma]\!](\overline{\mathcal{D}^{\dagger}}[\![(\eta)[\iota]\Delta]\!]\langle\rho,\sigma\rangle)\downarrow\rho.$$

Note that this fails (strictly speaking) to be a denotational description of the meaning of a block, because the right-hand side involves the meanings not of syntactic subterms of the block but of substituted versions thereof.

The following properties are inherited from the standard semantics.

Lemma 3. For all frames $\langle\rho,\sigma\rangle$, and for all Γ,

$$\overline{\mathcal{M}}[\![\Gamma]\!]\langle\rho,\sigma\rangle \equiv \langle\rho,\sigma\rangle \quad \text{on } \mathrm{dom}(\rho) - \rho(\mathrm{free}(\Gamma)). \quad \blacksquare$$

Theorem 4.

$$\langle\rho,\sigma\rangle \equiv_{\mathrm{free}(\Gamma)} \langle\rho',\sigma'\rangle \quad \Rightarrow \quad \overline{\mathcal{M}}[\![\Gamma]\!]\langle\rho,\sigma\rangle \equiv_{\mathrm{free}(\Gamma)} \overline{\mathcal{M}}[\![\Gamma]\!]\langle\rho',\sigma'\rangle.$$

Lemma 4. For all Δ, all Γ, and all frames $\langle\rho,\sigma\rangle$, the valuations

$$\overline{\mathcal{M}}[\![\mathbf{begin}\ \Delta;\ \Gamma\ \mathbf{end}]\!]\langle\rho,\sigma\rangle \quad \text{and} \quad \overline{\mathcal{M}}[\![\Gamma]\!](\overline{\mathcal{D}}[\![\Delta]\!]\langle\rho,\sigma\rangle)$$

agree on all identifiers $I \notin \mathrm{dec}[\![\Delta]\!]$. $\blacksquare$

Programs. We defined the abstract semantics of commands to be:

$$\overline{\mathcal{P}}[\![\mathbf{begin}\ \Delta;\ \Gamma;\ \mathbf{result}\ E\ \mathbf{end}]\!] \ = \ \overline{\mathcal{E}}[\![E]\!](\overline{\mathcal{M}}[\![\Gamma]\!](\overline{\mathcal{D}}[\![\Delta]\!]\langle\emptyset,\emptyset\rangle)).$$

For example, the program

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

has result 2.

By our previous results, it is evident that $\overline{\mathcal{P}} = \mathcal{P}$: all programs have the same results in the standard and abstract versions of the semantics.

Theorem. For all programs Π, $\overline{\mathcal{P}}[\![\Pi]\!] \ = \ \mathcal{P}[\![\Pi]\!]$.

Proof. The configuration $\langle\emptyset,\emptyset\rangle$ determines the empty sharing relation and the empty valuation. We have

$$\begin{aligned}
\overline{\mathcal{P}}[\![\mathbf{begin}\ \Delta;\ \Gamma;\ \mathbf{result}\ E\ \mathbf{end}]\!] &= \overline{\mathcal{E}}[\![E]\!](\overline{\mathcal{M}}[\![\Gamma]\!](\overline{\mathcal{D}}[\![\Delta]\!]\langle\emptyset,\emptyset\rangle)) \\
&= \overline{\mathcal{E}}[\![E]\!](\overline{\mathcal{M}[\![\Gamma]\!](\mathcal{D}[\![\Delta]\!]\langle\emptyset,\emptyset\rangle)}) \\
&= \overline{\mathcal{E}}[\![E]\!](\overline{\mathcal{M}[\![\Gamma]\!](\mathcal{D}[\![\Delta]\!]\langle\emptyset,\emptyset\rangle)}) \\
&= \mathcal{E}[\![E]\!](\mathcal{M}[\![\Gamma]\!](\mathcal{D}[\![\Delta]\!]\langle\emptyset,\emptyset\rangle)) \\
&= \mathcal{P}[\![\mathbf{begin}\ \Delta;\ \Gamma;\ \mathbf{result}\ E\ \mathbf{end}]\!].
\end{aligned}$$

4. Full Abstraction.

In this section we prove that the sharing class semantics is fully abstract with respect to the notion of program behaviour represented by $\mathcal{P}$. We also mention briefly what the corresponding results are for the location semantics given earlier.

We define the usual *semantic equivalence* relations as follows. Two commands are semantically equivalent iff they denote the same value:

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

Implicit in this is the requirement that the two commands have the same set of free identifiers. Thus, two commands are identified by the semantics iff whenever executed from the same frame they produce the same valuation: for all frames $\langle\rho,\sigma\rangle$, $\overline{\mathcal{M}}[\![\Gamma_0]\!]\langle\rho,\sigma\rangle = \overline{\mathcal{M}}[\![\Gamma_1]\!]\langle\rho,\sigma\rangle$.

Similarly, for the other syntactic categories we can define

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

Two expressions are equivalent iff they have the same free identifiers and always evaluate to the same value: for all frames $\langle\rho,\sigma\rangle$, $\overline{\mathcal{E}}[\![E_0]\!]\langle\rho,\sigma\rangle = \overline{\mathcal{E}}[\![E_1]\!]\langle\rho,\sigma\rangle$. And two declarations are equivalent iff they have the same free identifiers, the same declared identifiers, and they always produce the same effect: for all frames f, $\overline{\mathcal{D}}[\![\Delta_0]\!]f = \overline{\mathcal{D}}[\![\Delta_1]\!]f$. 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

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

Two programs are equivalent iff their results are the same.

It should be evident that $\mathcal{E}$ and $\overline{\mathcal{E}}$ determine exactly the same semantic equivalence, in that

$$\forall E_1, E_2.[\mathcal{E}[\![E_1]\!] = \mathcal{E}[\![E_2]\!] \quad \Leftrightarrow \quad \overline{\mathcal{E}}[\![E_1]\!] = \overline{\mathcal{E}}[\![E_2]\!]].$$

The same is almost true of the pair $\mathcal{M}$ and $\overline{\mathcal{M}}$, the only difficulty being that because of the fact that we allowed configurations containing inaccessible locations but block exit causes deallocation, the $\mathcal{M}$ semantics can distinguish between commands such as **skip** and **begin** Δ; **skip**, whereas these are identified in the $\overline{\mathcal{M}}$ semantics. No matter what decision is made about storage management, as discussed before, we get

$$\forall \Gamma_1, \Gamma_2.[\mathcal{M}[\![\Gamma_1]\!] \approx \mathcal{M}[\![\Gamma_2]\!] \quad \Leftrightarrow \quad \overline{\mathcal{M}}[\![\Gamma_1]\!] = \overline{\mathcal{M}}[\![\Gamma_2]\!]].$$

Similarly, again because of the storage issues,

$$\forall \Delta_1, \Delta_2.[\mathcal{D}[\![\Delta_1]\!] \approx \mathcal{D}[\![\Delta_2]\!] \quad \Leftrightarrow \quad \overline{\mathcal{D}}[\![\Delta_1]\!] = \overline{\mathcal{D}}[\![\Delta_2]\!]].$$

From now on we concentrate on the abstract semantics.

Clearly, each of these semantic 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. We will use the notation $\Pi[\,\cdot\,]$ for a program context, with the type being inferrable from the usage. We also use the notation $\Pi[t]$ for the result of filling the hole of a context with a term t 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? We devote the rest of this section to proving this. These full abstraction results depends on a simple *expressivity* property of the expression language **Exp** (*Assumption 2*):

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

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 *finitely many* identifiers.

Lemma 5. For any frame f there is a closed declaration Δ_f that *defines* f, in the sense that

$$\overline{\mathcal{D}}[\![\Delta_f]\!]\langle\emptyset,\emptyset\rangle = f.$$

Proof. By induction on the size of the set $\mathrm{dom}(f)$.

- If $\mathrm{dom}(f) = \emptyset$, we have $f = \langle\emptyset,\emptyset\rangle$ and we may simply put $\Delta_f = \mathbf{null}$.
- Otherwise, if $\mathrm{dom}(f)$ is non-empty, let $I_1 \in \mathrm{dom}(f)$. Let $f = \langle\rho,\sigma\rangle$, and let $\rho(I_1) = X = \{I_1, I_2, \ldots, I_k\}$. Let $v = \sigma(I_1)$. Let $\langle\rho_1,\sigma_1\rangle = \langle\rho|X, \sigma\backslash X\rangle$, so that $\rho = (\rho_1 \cup X^2)$ and $\sigma = \sigma_1 + [X \mapsto v]$. Clearly, $\langle\rho_1,\sigma_1\rangle$ is a frame defined on $A - X$, a set with smaller size than A. By the induction hypothesis we may choose a closed declaration Δ_1 defining f_1, i.e. so that

$$\overline{\mathcal{D}}[\![\Delta_1]\!]\langle\emptyset,\emptyset\rangle = \langle\rho_1,\sigma_1\rangle.$$

By Assumption 2, there is a closed expression E_v with value v. We may then choose Δ_f to be:

$$\begin{aligned}
\Delta_f \ &= \Delta_0;\ \Delta_1,\\
\text{where}\quad \Delta_0 \ &= \mathbf{new}\ I_1 = E_v;\ \mathbf{alias}\ I_2 = I_1; \ldots;\ \mathbf{alias}\ I_k = I_1.
\end{aligned}$$

Clearly, Δ_0 is also closed, and its effect is to place the identifiers $I_1, \ldots, I_k$ into a single sharing class initialized to the value v:

$$\overline{\mathcal{D}}[\![\Delta_0]\!]\langle\emptyset,\emptyset\rangle \;=\; \langle X^2, [X \mapsto v]\rangle.$$

Since both Δ_0 and Δ_1 are closed, so is Δ_f. Since $\mathrm{dec}(\Delta_0) \cap \mathrm{dec}(\Delta_1) = \emptyset$ and Δ_1 is closed (so that $\alpha(\Delta_1) = \emptyset$),

$$\begin{aligned}\overline{\mathcal{D}}[\![\Delta_0;\ \Delta_1]\!]\langle\emptyset,\emptyset\rangle &= \langle \rho_1 \cup X^2, \sigma_1 + [X \mapsto v]\rangle \\ &= \langle \rho, \sigma\rangle,\end{aligned}$$

as required.

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

Lemma 6. For any frame $f = \langle\rho,\sigma\rangle$, and any finite set $A \subseteq \mathrm{dom}(\rho)$, there is a command Γ_f^A such that for all σ' consistent with ρ the valuation of $\overline{\mathcal{M}}[\![\Gamma_f^A]\!]\langle\rho,\sigma'\rangle$ agrees with σ on A.

Proof. • If $A = \emptyset$ put $\Gamma_f^A = \mathbf{skip}$.

• If A is not empty, let $I \in A$ and $B = A - \rho(I)$. Let $v = \sigma(I)$. By inductive hypothesis, there is a command Γ_f^B such that whenever σ' is consistent with ρ, the valuation $\overline{\mathcal{S}^\dagger}[\![\Gamma_f^B]\!]\langle\rho,\sigma'\rangle$ agrees with σ on B. This valuation is also consistent with ρ. There is a closed expression E_v with value v. We may therefore put

$$\Gamma_f^A \;=\; \Gamma_f^B;\ I{:=}E_v,$$

so that whenever σ' is consistent with ρ we will get

$$\begin{aligned}\overline{\mathcal{S}^\dagger}[\![\Gamma_f^A]\!]\langle\rho,\sigma'\rangle &= \overline{\mathcal{S}^\dagger}[\![I{:=}E_v]\!]\langle\rho, \overline{\mathcal{S}^\dagger}[\![\Gamma_f^B]\!]\langle\rho,\sigma'\rangle\rangle \\ &= \overline{\mathcal{S}^\dagger}[\![\Gamma_f^B]\!]\langle\rho,\sigma'\rangle + [\rho(I) \mapsto v].\end{aligned}$$

This valuation agrees with σ on A, as required.

∎

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

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

Proof.

• For $\overline{\mathcal{E}}$, we wish to show that for all expressions E_0 and E_1,

$$\forall\Pi[\,\cdot\,].(\Pi[E_0] \equiv \Pi[E_1]) \quad\Rightarrow\quad \mathcal{E}[\![E_0]\!] \;=\; \mathcal{E}[\![E_1]\!].$$

This is easy to prove. If $\mathcal{E}[\![E_0]\!] \neq \mathcal{E}[\![E_1]\!]$, there is a frame $f = \langle\rho,\sigma\rangle$ such that $\mathcal{E}[\![E_0]\!]\sigma$ is different from $\mathcal{E}[\![E_1]\!]\sigma$. Choose a closed declaration Δ_f defining f as in Lemma 5. The program context

$$\mathbf{begin}\ \Delta_f;\ \mathbf{skip};\ \mathbf{result}\ [\,\cdot\,]\ \mathbf{end}$$

will distinguish between E_0 and E_1.

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

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

Suppose that $\overline{\mathcal{D}}[\![\Delta_0]\!] \neq \overline{\mathcal{D}}[\![\Delta_1]\!]$. We will construct a program context to distinguish between these two declarations. By assumption, there is a frame f and an identifier I such that

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

Let $\overline{\mathcal{D}}[\![\Delta_i]\!]f = \langle \rho_i, \sigma_i \rangle = f_i$, for $i = 0, 1$. We know that either the *values* $\sigma_i(I)$ differ, or the *sharing classes* $\rho_i(I)$ differ. There are thus two cases to consider.

Firstly, if the value of I is different in f_0 and f_1, we may choose using Lemma 5 a closed declaration Δ_f defining f as above. Then the program context

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

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. By Lemma 5 there is a declaration Δ_f defining f. The program context

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

will distinguish between Δ_0 and Δ_1.

• For $\overline{\mathcal{M}}$ a similar argument can be based on Lemmas 5 or on Lemma 6. Here is a proof using Lemma 5. A different distinguishing context can be built based on Lemma 6. If $\overline{\mathcal{M}}[\![\Gamma_0]\!]$ and $\overline{\mathcal{M}}[\![\Gamma_1]\!]$ differ, there is a frame f and an identifier I such that

$$(\overline{\mathcal{M}}[\![\Gamma_0]\!]f)(I) \neq (\overline{\mathcal{M}}[\![\Gamma_1]\!]f)(I).$$

Choose a closed declaration Δ_f defining f as in Lemma 5. The context

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

distinguishes between Γ_0 and Γ_1.

∎

Location Semantics.

We have proved that the sharing relation semantics is fully abstract with respect to the behavioural equivalence corresponding to $\mathcal{P}$. Since we have indicated that the location semantics failed in this respect, it is worth noting here briefly that nevertheless the location semantics is still fairly close to being fully abstract. These properties are immediate corollaries of previously stated results.

$$\begin{aligned}
\forall \Pi[\,\cdot\,](\mathcal{P}[\![\Pi[E_1]]\!] = \mathcal{P}[\![\Pi[E_2]]\!]) \quad &\Leftrightarrow \quad \mathcal{E}[\![E_1]\!] = \mathcal{E}[\![E_2]\!], \\
\forall \Pi[\,\cdot\,](\mathcal{P}[\![\Pi[\Gamma_1]]\!] = \mathcal{P}[\![\Pi[\Gamma_2]]\!]) \quad &\Leftrightarrow \quad \mathcal{M}[\![\Gamma_1]\!] \approx \mathcal{M}[\![\Gamma_2]\!], \\
\forall \Pi[\,\cdot\,](\mathcal{P}[\![\Pi[\Delta_1]]\!] = \mathcal{P}[\![\Pi[\Delta_2]]\!]) \quad &\Leftrightarrow \quad \mathcal{D}[\![\Delta_1]\!] \approx \mathcal{D}[\![\Delta_2]\!].
\end{aligned}$$

Now we return to the sharing class semantics. Having shown that it is fully abstract, we now build an axiomatic treatment of program properties based closely on the semantic model.

5. Axiomatic Semantics.

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 relative completeness of the proof system are easy (if tedious) 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 this programming language, declarations have effects on both the sharing relation (a *declarative* effect) and on the association of identifiers to values (an *imperative* effect), because of the initializations 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 (except locally, during block execution). 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 determined by the set of declarations in whose scope a command is executing. There is a reasonably obvious notion of when a declaration Δ specifies that the set X of identifiers is a sharing class. We may formalize this notion precisely. 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 relation, 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 have a finite domain, 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 the Y_j list is simply a re-ordering of the X_i list with the possible inclusion of some extra trivial (empty) classes. The interpretation of such an assertion is clear: 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 $X_1, \ldots, X_n$ and I belongs to X_i, then $\phi(I) = X_i$; if I is not included in any of the listed classes, then

$\phi(I) = \emptyset$. We also introduce the notation $\phi - I$ for the result of removing I from every sharing class in the list ϕ. And we use $\phi \backslash I$ for the result of deleting the sharing class $\phi(I)$ from the list ϕ.

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 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:

$$\langle \phi \rangle \mathbf{new}\ I = E \langle \{I\}, \phi - I \rangle \tag{A2}$$

• 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:

$$\langle \phi \rangle \mathbf{alias}\ I_0 = I_1 \langle \phi(I_1) \cup \{I_0\}, (\phi \backslash I_1) - I_0 \rangle \tag{A3}$$

• 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

$$\begin{aligned}
&\langle\rangle \mathbf{new}\ x = 0 \langle \{x\} \rangle \\
&\langle \{x\} \rangle \mathbf{alias}\ y = x \langle \{x, y\} \rangle \\
&\langle \{x, y\} \rangle \mathbf{alias}\ z = w \langle \{x, y\}, \{z, w\} \rangle \\
&\langle \{x\} \rangle \mathbf{alias}\ y = x;\ \mathbf{alias}\ z = y \langle \{x, y, z\} \rangle.
\end{aligned}$$

It should be clear that these axioms correspond very closely to the semantic function $\mathcal{R}$. Indeed, it is easy to formalize 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

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

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

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

This proof system is sound and complete. The proofs are almost trivial, and we relegate them to the Appendix.

Theorem 7. (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 8. (Completeness) For all Δ and all ϕ, ψ,

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

Finally, note that with our choice of assertion language we have the following important properties, which state precisely that the assertion language is as discriminating as necessary over the semantic structure in which it is being interpreted. Firstly, every sharing relation ρ satisfies some assertion ϕ; and, secondly, for every pair $\rho_1 \neq \rho_2$ of distinct sharing relations, there is an assertion ψ satisfied by one but not the other, i.e. such that $\rho_1 \models \psi$ but $\rho_2 \not\models \psi$. As a consequence, it is easy to show that the assertions about declarations are as discriminating as necessary:

Theorem 9. For all Δ_1 and Δ_2, $\overline{\mathcal{R}^\dagger}[\![\Delta_1]\!] \neq \overline{\mathcal{R}^\dagger}[\![\Delta_2]\!]$ if and only if there are ϕ and ψ such that

$$\models \langle\phi\rangle\Delta_1\langle\psi\rangle \quad \text{but} \quad \not\models \langle\phi\rangle\Delta_2\langle\psi\rangle.$$

In other words, adapting the terminology of Meyer and Halpern [21], "the set of valid assertions (about declarations) defines the (declarative) semantics ($\overline{\mathcal{R}^\dagger}$)".

Imperative Proof System.

None of the assertions used above 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 formulate proof rules for commands. Essentially, we are separating entirely the purely binding effect of a declaration from the initialization 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. 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. For concreteness, we will assume that the condition language includes atomic conditions such as $I = E$ and is closed under the usual logical connectives.

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 (if the computation terminates) 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 are led to a natural generalization of these 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 (provided the computation terminates) the final valuation will satisfy Q. Note that the structure of our generalized assertion matches the type of the semantic function $\overline{\mathcal{M}}$.

For declarations, we observed that the (local) imperative effect of a declaration, described by the semantic function $\mathcal{S}$, 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.

For the imperative effects of declarations, we provide the following rules.

- 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 initializing effect is similar:

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

- Sequential composition of declarations behaves simply:

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

Now we give the clauses for commands. As usual, we give a clause for each command construct. By making use of the prior axiomatization of the semantics of declarations we are able to obtain a rather simple treatment for blocks.

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

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

• 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 generalization 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 soundness of this rule relies on a standard property of syntactic substitution. The statement of this property for our expression language is that for all valuations σ and all $Y \subseteq \mathbf{Ide}$ we have

$$\mathcal{E}[\![[E\backslash Y]E']\!]\sigma \;=\; \mathcal{E}[\![E']\!](\sigma + [Y \mapsto \mathcal{E}[\![E]\!]\sigma]). \tag{SUB}$$

• 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 declaration we have to bear in mind 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. Note that the premisses of the rule involve assertions about both the imperative and declarative effect of the declaration at the head of the block. It is here more than anywhere that the separate axiomatization of declarative semantics is helpful. The rule is sound provided none of the declared identifiers in Δ occurs free in R:

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

The need for the syntactic constraints was mentioned earlier in the statement of Lemma 4, which guarantees the soundness of this rule.

So far we do not have any rule corresponding to "change of bound variable." The block rule (B4) 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, and they may refer to different variables inside the block. We can suppress the need to reason explicitly about the $\mathcal{T}$ semantics by changing bound identifiers to avoid hole-in-scope problems. We need, therefore, to be able to deduce 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 to our setting of standard rules from the literature (see [1,2] for example). Its soundness relies on the Substitution Properties established earlier. The rule is:

$$\frac{\phi \vdash \{P\}\mathbf{begin}\ (\eta)[\iota]\Delta;\ [\iota + \eta]\Gamma\ \mathbf{end}\{R\}}{\phi \vdash \{P\}\mathbf{begin}\ \Delta;\ \Gamma\ \mathbf{end}\{R\}} \tag{B9}$$

when η is a substitution on $\mathrm{dec}(\Delta)$ with $rge(\eta)$ disjoint from $\mathrm{free}(R)$. The premiss of this inference rule involves a block in a form to which rule (B4) may be applied directly.

In addition to the above syntactically motivated rules, the utility and necessity of the following rule should be self-evident. It allows us to use the consistency property of frames 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 versions of the rule of consequence. Note that it is necessary to include the sharing assertion explicitly in the rule of consequence for commands.

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

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

Comparison with other proof rules.

Clearly, many of our proof rules and axioms have essentially the same form as well known rules in the literature. Indeed, in the absence of (non-trivial) sharing, our axioms and rules for commands collapse down to standard rules, as we would expect. In particular, the assignment rule collapses to Hoare's original axiom [16]:

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

and our rule for sequential composition of commands collapses to Hoare's rule. The main differences are evident in the treatment of aliasing relationships. By axiomatizing the properties of declarations, we were able to build a rule for assignment which incorporates explicit reasoning about sharing and generalizes Hoare's assignment rule in the obvious simple manner, replacing a single substitution by a multiple simultaneous substitution. A similar rule for a multiple assignment statement appears in [11,12], although aliasing was not considered there.

A further benefit of our prior axiomatization of the effect of declarations on the sharing relation is that we are able to design simple proof rules for blocks. Our rule (B4) for blocks can be said to be more truly syntax-directed than is usual in the literature, because the rule as given here is in a "generic" form applicable to more than one form of declaration. The structure of the rule does not depend on the precise form of declaration with which the block begins. The following specialized 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 [1,2,4,5], although

direct comparison is somewhat hampered by the differences in syntax. In these references, in particular, *new* declarations do not also initialize the value of the declared identifier to the value of an expression. Aliasing is not treated in [1,2], so that the block rule stated there is interpreted in a sharing-free setting.

A rule for a block beginning with a *new* declaration can be obtained from rules (A2), (B4) and (B9), together with rule (B6). Firstly, note that if I does not occur free in E and P, then the following will be provable as a special case of (B6) after an application of the rule of consequence:

$$\{P\}\mathbf{new}\ I = E\{P\ \&\ I = E\}.$$

Therefore, if I does not occur free in P, E, or R, we can use rules (B4) and (A2) to derive the rule:

$$\frac{\phi - I, \{I\} \vdash \{P\ \&\ I = E\}\Gamma\{R\}}{\phi \vdash \{P\}\mathbf{begin\ new}\ I = E;\ \Gamma\{R\}} \tag{D1}$$

If we also make use of the change of bound variables rule to ensure the necessary constraints, we obtain the following derived rule. If I' is a fresh identifier which does not occur free in Γ, E, P, or R, then

$$\frac{\phi - I', \{I'\} \vdash \{P\ \&\ I' = E\}[I'\backslash I]\Gamma\{R\}}{\phi \vdash \{P\}\mathbf{begin\ new}\ I = E;\ \Gamma\ \mathbf{end}\{R\}}. \tag{D2}$$

In this form, the rule is a generalization of the Variable Declaration Rule (Rule 16) of [1], with suitable modification to allow for the initialization effect of a declaration and for the sharing assumption.

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

$$\frac{\langle\phi\rangle\mathbf{alias}\ I_0 = I_1\langle\psi\rangle \quad \psi \vdash \{P\ \&\ I_0 = I_1\}\Gamma\{R\}}{\phi \vdash \{P\}\mathbf{begin\ alias}\ I_0 = I_1;\ \Gamma\ \mathbf{end}\{R\}}, \tag{D3}$$

provided I_0 does not occur free in P, Q, Γ. Again, a version incorporating a change of bound variable can also be formulated, and an explicit representation for ψ can be derived from rule (A3).

An important point to note here is that these block rules from the literature turn up as *derived* rules and not primitive rules in our system. This suggests, as we claimed earlier, that our formulation of the proof system is more truly syntax-directed. We would be able to add new forms of declaration to the language, and to add new axioms and rules for these extra constructs, without modifying the structure of the proof rule for blocks (provided, of course, that the additional constructs can be semantically modelled inside our current framework); we would not have to introduce a separate proof rule for blocks beginning with each different form of declaration, although of course specialized versions of the block rule would be derivable. It is this adaptability of our proof system that we regard as an important asset.

In the language under consideration in this paper, declarations have both imperative and declarative aspects, and we were able to focus on these aspects independently. If the programming language only allowed purely declarative declarations which do not have any non-trivial initializing effect (for example an uninitialized declaration **new** x), we would not need any assertions of the form $\{P\}\Delta\{Q\}$; our proof system could then be adapted to this setting in an obvious way. Conversely, if we add declarations whose declarative effect depends on the valuation as well as on the sharing relation, the imperative assertions

for declarations would need to be made more complicated. This would be the case, for instance, if we allowed array declarations, so that one could then declare **alias** $x = a[y]$. The declaration-time value of y would then determine the sharing class of x. One solution here would be to use assertions of the form $\phi \vdash \{P\}\Delta\langle\psi\rangle$, or some alternative sugared form; in any case, the assertion would need some subterm to represent an assumption about the initial valuation. The point is that we believe that our general methods will be applicable in a wider setting than the one considered in this paper, provided we are ready to adopt the use of assertions whose structure more closely follows the semantic structure of the programming language under examination.

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

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

by (B6) and (B12). We also have, by (A2),

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

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

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

by the assignment rule (B2). Similarly,

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

Applying rule (B3) for sequential composition, we get

$$\{x = 0 \ \& \ y = 1 \ \& \ t = 0\}x{:=}y; \ y{:=}t\{x = 1 \ \& \ y = 0 \ \& \ t = 0\}.$$

The result follows by the block rule (B4) and the rule of consequence (B12).

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 *alias* and sequential composition of 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 $w \notin X$. 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 relatively 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. We have already given a satisfaction relation $\models$ for sharing relations and sharing assertions. We also need a satisfaction relation for valuations and conditions. Although we have not been specific about the syntax of the condition language, it is reasonable to assume that we are given a satisfaction relation $\sigma \models P$, which behaves correctly with respect to the usual logical connectives and with respect to syntactic substitution. By this we mean that for all conditions P, all $Y \subseteq \mathbf{Ide}$ and all $E \in \mathbf{Exp}$ we have

$$\sigma \models [E\backslash Y]P \quad \Leftrightarrow \quad (\sigma + [Y \mapsto \overline{\mathcal{E}}[\![E]\!]\sigma]) \models P.$$

These are standard assumptions which are usually made tacitly in axiomatic treatments. The above requirement on substitution is analogous to the property (SUB) mentioned earlier.

We also define a composite notion of satisfaction by:

$$\langle\rho,\sigma\rangle \models \langle\phi, P\rangle \quad \Leftrightarrow \quad (\rho \models \phi)\ \&\ (\sigma \models P).$$

We say that an assertion

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

is *valid*, denoted

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

iff whenever $\langle\rho,\sigma\rangle$ satisfies $\langle\phi, P\rangle$ then $\overline{\mathcal{M}}[\![\Gamma]\!]\langle\rho,\sigma\rangle$ satisfies $\langle\phi, Q\rangle$: for all $\langle\rho,\sigma\rangle$,

$$\langle u, s\rangle \models \langle\phi, P\rangle \quad \Rightarrow \quad \overline{\mathcal{M}}[\![\Gamma]\!]\langle\rho,\sigma\rangle \models \langle\phi, Q\rangle.$$

Similarly, an assertion $\{P\}\Delta\{Q\}$ is valid if whenever σ satisfies P then $\overline{\mathcal{S}^\dagger}[\![\Delta]\!]\sigma$ satisfies Q: for all σ,

$$\sigma \models P \quad \Rightarrow \quad \overline{\mathcal{S}^\dagger}[\![\Delta]\!]\sigma \ \models\ Q.$$

We need to show that all valid assertions are provable, and every provable assertion is valid. As usual, following Cook [7], 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:

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

We want to prove that all valid assertions are provable *relative to* **Th**, the standard notion of *relative completeness.*

The proofs of soundness are straightforward. 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 10.(Soundness) 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 11.(Soundness) 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 rule (B4) follows from Lemma 4. The soundness of the rules for sequential composition of declarations and commands follows directly from the semantic definitions. The consistency rule (B10) is sound because it can be proved by structural induction that the semantic definitions preserve consistency of the valuation with respect to the sharing relation.

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 12.(Completeness) 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 13.(Completeness) For all Γ and for all ϕ, P and Q,

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

Although we did not fix the syntax of the condition language we did assume that it contained atomic conditions such as $I = E$ and that it possessed the usual logical connectives. It follows that for every finite set of identifiers and every valuation defined on that set we can find a *characteristic condition*: for every σ and every finite set of identifiers A there is a condition P such that for all σ', $\sigma' \models P$ if and only if σ and σ' agree on A. Hence, for every pair of distinct valuations on the same identifiers there is a condition true of one but not the other, i.e., for every $\sigma_1 \neq \sigma_2$ there is a Q such that $\sigma_1 \models Q$ and $\sigma_2 \not\models Q$. Using these properties, the following results are obtainable:

Theorem 14. For all Δ_1 and Δ_2, $\overline{\mathcal{S}}[\![\Delta_1]\!] \neq \overline{\mathcal{S}}[\![\Delta_2]\!]$ (equivalently, $\overline{\mathcal{S}^\dagger}[\![\Delta_1]\!] \neq \overline{\mathcal{S}^\dagger}[\![\Delta_2]\!]$) if and only if there are conditions P and Q such that

$$\models \{P\}\Delta_1\{Q\} \quad \text{but} \quad \not\models \{P\}\Delta_2\{Q\}. \quad \blacksquare$$

Theorem 15. For all Γ_1 and Γ_2, $\mathcal{M}[\![\Gamma_1]\!] \neq \mathcal{M}[\![\Gamma_2]\!]$ if and only if there are ϕ, P, and Q such that

$$\models (\phi \vdash \{P\}\Gamma_1\{Q\}) \quad \text{but} \quad \not\models (\phi \vdash \{P\}\Gamma_2\{Q\}). \quad \blacksquare$$

These results state precisely our claim that the semantics defined in the first part of the paper does indeed identify terms if and only if they satisfy exactly the same partial correctness assertions. The set of valid imperative assertions about declarations defines $\mathcal{S}$, and the set of valid assertions about commands defines $\mathcal{M}$, in the sense of [21].

6. Conclusions.

The work reported in this paper is an attempt to design a clean and mathematically tractable semantics for a programming language, a semantics which is specifically intended to serve as a basis for reasoning about a particular type of program behaviour: in this case partial correctness. We considered a simple programming language with block structure and a form of aliasing, so that we were able to concentrate on the problems inherent in these features alone. The underlying semantic model with respect to which we proved soundness and completeness was location-free, and we proved its suitability as the basis for formal reasoning about partial correctness by demonstrating that the semantics was fully abstract with respect to a related notion of program behaviour. We further demonstrate this suitability by using the semantics directly in the design of an assertion language and proof system for reasoning about partial correctness of programs. The semantics may be used to prove the soundness and relative completeness of this proof system, and as a consequence of the choice of semantic model these proofs may be formulated rather cleanly. We were also able to derive some well known proof rules from the literature.

As we stated earlier, we intended in this paper to focus on a small number of programming language features and to concentrate exclusively on the problems caused by aliasing. The programming language consequently omitted many features which would be required in any more realistic setting. With minor modifications we can add loops and conditionals; the only essential difference is that we then need to justify the use of recursive definitions in the semantic description of loops. This is straightforward if we impose a natural partial ordering on the relevant semantic structure and follow the standard lines of the Scott-Strachey approach. We can still obtain a full abstraction result for the enlarged language and its semantics. Hoare's proof rules for loops and conditional commands can be adapted in the obvious way to this setting, by incorporating a sharing assertion. Similarly, although this paper ignored the possibility of run-time errors such as an attempt to evaluate or assign to an undeclared or uninitialized identifier, it is easy to modify the syntax and semantics of the language in order to cope with these problems.

It is well known that block-structured languages with static scope rules can be implemented with stacks, and several semantic treatments have been suggested which incorporate such techniques, notably by Landin [19], Jones [18], Olderog [25,26] and Langmaack [20]. Indeed, our treatment can be rewritten to make the stack discipline explicit. Landin used an abstract machine with a "sharing component" which explicitly described the sharing relationships among program variables, and gave operational semantics to a variety of programming languages. Jones also tried to use sharing relations and consistent valuations, in an operational semantic framework. Langmaack and Olderog give axiomatic treatments of aliasing which involve a notion of sharing classes, specifically to deal with sharing among actual parameters in procedure calls. They did not give a separate denotational description of sharing classes and their manipulation during program execution, and their notion of stacks seems to involve locations rather more explicitly. We feel that our treatment is somewhat cleaner, and gains in simplicity and clarity by explicitly focussing on the need to describe separately the denotational semantics of declarations; nevertheless, we should also admit that some of the extra clarity is obtained because we are working with a less complicated programming language.

An approach to the semantics of block-structured languages using category theory has been developed by Oles and Reynolds [27], and there are connections between their abstract store models and our sharing class model. Our frames and links can be regarded as the objects and arrows of a category, and, as we observed in passing earlier, this is related

to the category of *store shapes* introduced by Oles and Reynolds. Further influences of the work of Reynolds are visible in some of our semantic results which were used in the full abstraction proof. Our Theorems 1–4 can be seen as a rigorous analysis of the relevant notions of *interference* for our programming language [30,31,32,36], since they state precisely the conditions under which a declaration or command can affect or be affected by the value of identifiers.

Other related work includes the proof system of [13] and the semantic treatments of [14,15], based on a rather intricate notion of "store-model"; in contrast to the approach used there, our underlying semantic model is arguably cleaner and we have proved full abstraction, albeit for a smaller language. We feel that our proof rules are more natural, although again some of this "benefit" arises because we have a simpler programming language, and it remains to be seen what happens when we extend our methods to a language with procedures of higher type.

We believe that many existing programming language features still lack elegant and tractable formal description, and that their axiomatization has been attempted somewhat prematurely, with insufficient attention to semantic issues. Full abstraction gives one criterion for judging the suitability of a semantics for supporting formal reasoning about program behaviour; simplicity of semantic structure gives another (admittedly more subjective). The construction of fully abstract semantics for many programming languages, even with respect to such well known behavioural criteria as partial correctness, has not yet been adequately investigated (notable exceptions being [23,28,29]). Indeed, it seems likely that full abstraction and simple structure are difficult and may often be impossible to achieve simultaneously. However, it should be noted that our methodology does not depend on the achievement of full abstraction; rather, it is crucial for us to begin with a semantics structure as simple as possible while still powerful enough to support proper treatment of program properties. Even in the absence of full abstraction, a clean semantic structure is advantageous if one wants to construct semantically based proof systems.

An important suggestion illustrated by 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. Similar observations have been made by other authors, notably by Sokolowski (with respect to expressions) in [33], by Goerdt in [9] (typed α-terms), and by Boehm in [3] (expressions with side-effects). Indeed, Boehm describes an axiomatic treatment of aliasing, again involving explicit reasoning about locations. Our decision to separate the different semantic aspects of the programming language and to build a hierarchical proof system whose structure reflects the syntactic and semantic structure of the language has been influenced by some of the ideas in Reynolds' *specification logic*, although Reynolds points out in [31] that in its present state specification logic does not seem to support reasoning about call-by-reference (which is analogous to our form of alias declaration).

In principle, it should be possible to use a semantics directly 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, indeed, for a language with many syntactic classes, a richer notation would be needed for designating assertions about the various classes of terms); and the axioms and rules may not be as clean as the ones we were able to use here.

Our methods suggest, we feel, a general basis for constructing Hoare-like semantics for some more complicated languages involving sharing, such as languages including array declarations and array assignments. Although we have not worked out the details, our approach here would lead to proof rules involving explicit reasoning about the sharing classes of array expressions, unlike the rules of [2], which used a generalization of syntactic substitution to cope with array assignments. It should also be possible to treat more complex forms of sharing, such as the hierarchical sharing structures required to deal with ALGOL 68 *ref* declarations and pointers.

Adding procedures with various forms of parameter passing is a much more interesting and difficult problem. It is known that procedure calls in statically scoped languages can be implemented by a form of copy rule, replacing a procedure call by a block containing a declaration which binds formal parameters to actual parameters (see, for example, Tennent[37]). Our simple programming language contained two types of declaration, chosen to correspond to two parameter mechanisms (we might use the terminology "call-by-alias", corresponding to call-by-reference (PASCAL **var** parameters), and "call-by-new", similar to PASCAL call-by-value). If we wish to add explicit procedures, by means of a procedure declaration and call, we might add to the syntax the clauses:

$$\begin{aligned} \Delta &::= \textbf{proc } P(\textbf{new } I) = \Gamma \mid \textbf{proc } P(\textbf{alias } I) = \Gamma \\ \Gamma &::= P(E) \mid P(I). \end{aligned}$$

It would then be possible to mimic the style of proof systems such as [25,26] in which the proof rules for procedures are based on copy rules. In future work, we plan to investigate this, as well as the use of a more explicitly semantically based assertion structure; since procedures denote functions from argument values to command values, a semantically based assertion language should use assertions which incorporate an appropriate type of "parameter condition". An assertion about (for example) a procedure having a single call-by-alias parameter should contain a parameter condition describing a sharing class. Some results relevant to this have been discussed by Reynolds [30,31] in his Specification Logic. This, and the consideration of further problematic features associated with procedures, such as recursion and the use of procedures as parameters, are topics for future research.

In summary, the adoption of a more widely based notion of Hoare system, with more attention to the semantic foundations, should lead to significant improvements in the axiomatic treatment of many programming language constructs. The use of semantics directly in the design and construction of assertion languages for reasoning about program properties should be advocated more extensively. Our work in this paper is a small step in this direction.

Acknowledgements. The author is grateful to Robert Cartwright, Joe Halpern, Albert Meyer, and Boris Trakhtenbrot for helpful comments and criticism, and to Allen Stoughton for pointing out an error in an earlier definition of $\mathcal{M}$. Suggestions by Paola Giannini, Ulrik Jørring, Brad White, and Bill Roscoe led to improvements in the presentation and development of the material. The referees provided helpful constructive criticism which has led to further clarification of the paper's material.

7. Appendix.

In this Appendix we sketch the completeness of our proof systems, introducing strongest post-conditions and weakest pre-conditions of appropriate types. The proof follows standard lines, exemplified by the proofs in [7,2].

Weakest preconditions and strongest postconditions.

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:

$$\begin{array}{rcl}
\mathrm{sp}[\![\textbf{new } I = E]\!](\phi) & = & \{I\}, \phi - I \\
\mathrm{sp}[\![\textbf{alias } I_0 = I_1]\!](\phi) & = & \phi(I_1) \cup \{I_0\}, (\phi \backslash I_1) - I_0 \\
\mathrm{sp}[\![\Delta_0; \Delta_1]\!](\phi) & = & \mathrm{sp}[\![\Delta_1]\!](\mathrm{sp}[\![\Delta_0]\!](\phi))
\end{array}$$

where we have used the notation $\phi \backslash I_1$ for the assertion obtained by omitting $\phi(I_1)$ from ϕ. 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,

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

The full property is expressed by the following lemma. Its proof is trivial.

Lemma C1. For all Δ, and all ϕ and ψ, $\models \langle\phi\rangle\Delta\langle\psi\rangle$ iff $\models (\mathrm{sp}[\![\Delta]\!](\phi) \Rightarrow \psi)$. ∎

Lemma C2. For all Δ, and all ϕ, $\vdash \langle\phi\rangle\Delta\langle\mathrm{sp}[\![\Delta]\!](\phi)\rangle$. ∎

Next, we define weakest pre-conditions.

For declarations we define

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

The following syntactic construction will then suffice for commands.

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

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

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

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

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

Lemma C3. For all Δ, and all P and Q,

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

Lemma C4. For all Γ, and all ϕ, P and Q,

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

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

Lemma C5. For all Δ, and all P and Q,

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

Lemma C6. For all Γ, and all ϕ, P and Q,

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

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

Theorem 13. 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 C6. 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.

8. 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] Boehm, H.-J., Side Effects and Aliasing Can Have Simple Axiomatic Descriptions, ACM TOPLAS, vol. 7, no. 4 (October 1985).

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

[5] Cartwright, R., and Oppen, D., Unrestricted Procedure Calls in Hoare's Logic, Proc. 5th ACM Symposium on Principles of Programming Languages, ACM New York.

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

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

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

[9] Goerdt, A., A Hoare Calculus for Functions Defined by Recursion on Higher Types, Proc. Logics of Programs, Brooklyn College, Springer Verlag LNCS 193, pp 106-117 (1985).

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

[11] Gries, D., The Multiple Assignment Statement, IEEE Trans. Software Engrg, SE-4, pp 89-93 (1978).

[12] Gries, D., and Levin, G., Assignment and procedure call proof rules, ACM TOPLAS 2, pp 564-579 (1980).

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

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

[15] 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).

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

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

[18] Jones, C. B., Yet Another Proof of the Correctness of Block Implementation, IBM Laboratory, Vienna (August 1970).

[19] Landin, P. J., A Correspondence between Algol 60 and Church's lambda notation, CACM 8, 89-101 and 158-165 (1965).

[20] Langmaack, H., On Termination Problems for Finitely Interpreted ALGOL-like Programs, Acta Informatica 18, pp 79-108 (1972).

[21] Meyer, A. R., and Halpern, J. Y., Axiomatic Definitions of Programming Languages: A Theoretical Assessment, JACM 29 (1982), pp 555-576.

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

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

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

[25] Olderog, E-R., Correctness of Programs with Pascal-like Procedures without Global Variables, Theoretical Computer Science 30 (1984) 49-90.

[26] Olderog, E-R., Sound and Complete Hoare-like Calculi Based on Copy Rules, Acta Informatica 16, pp 161-197 (1981).

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

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

[29] 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).

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

[31] Reynolds, J., Idealized Algol and its specification logic, Technical Report 1-81, Dept. of Computer and Information Science, Syracuse University (July 1981).

[32] Reynolds, J., Syntactic control of interference, Proceedings of the 5th ACM Symposium on Principles of Programming Languages, ACM New York, pp 39-46 (1978).

[33] Sokolowski, S., Partial correctness: The term-wise approach, Science of Computer Programming, vol. 2, no. 4, pp. 141-157, (August 1984).

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

[35] Strachey, C., Towards a formal semantics, in: Formal Language Description Languages for Computer Programming, ed. T. B. Steel, Jr., North-Holland, Amsterdam (1966).

[36] Tennent, R. D., Semantics of interference control, Theoretical Computer Science 27 (1983) 293-310.

[37] Tennent, R. D., Principles of Programming Languages, Prentice-Hall (1981).