Constraints:
A Uniform Approach to Aliasing and Typing

Leslie Lamport* — SRI International
Fred B. Schneider† — Cornell University

24 July 1984
revised 4 September 1984
minor revision 26 October 1984

Abstract

A constraint is a relation among program variables that is maintained throughout execution. Type declarations and a very general form of aliasing can be expressed as constraints. A proof system based upon the interpretation of Hoare triples as temporal logic formulas is given for reasoning about programs with constraints. The proof system is shown to be sound and relatively complete, and example program proofs are given.

1 Introduction

Type declarations and aliasing relations have traditionally been thought of as unrelated concepts. However, both can be viewed as specifying properties that do not change during program execution. This view leads to a uniform method for reasoning about types and aliasing in which the usual Hoare logic triples are regarded as temporal logic formulas.

Aliasing two variables x and y means they always have the same value. This is usually implemented by allocating the same memory location to x

*Work supported in part by the National Science Foundation under grant number MCS-8104459 and by the Army Research Office under grant number DAAG29-83-K-0119. Current address: SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025.

†Research supported in part by NSF grant DCR-8320274 and by a Faculty Development Award from IBM Corp. Current address: Department of Computer Science, Cornell University, Ithaca, NY 14853.

and y, thereby ensuring that both variables are changed whenever either is assigned a new value. However, they could be allocated separate memory locations and both updated on an assignment to either. Viewing aliasing as defining certain relationships between the values of variables, with no implication about storage allocation, allows more general kinds of aliasing and leads to a simple method for reasoning about aliasing.

To express a more general form of aliasing, we introduce the **var** statement. To illustrate its use, suppose a program computes a temperature, and that some times it is convenient to refer to that temperature in degrees Fahrenheit and other times in degrees Celsius. We will write the statement

$$\textbf{var}\ f, c : \mathit{real}\ \ \textbf{constraints}\ f = 9 * c/5 + 32\ \textbf{in}\ S$$

which declares variables f and c within statement S to be of type *real* and to be *aliased*, so that if the value of f is a temperature in degrees Fahrenheit, then the value of c is that temperature in degrees Celsius. Changing f causes a corresponding change to c, and vice-versa. Notice that this more general form of aliasing cannot be implemented simply by allocating overlapping memory locations to f and c.

The **constraints** clause of a **var** statement is a directive that a specified predicate—in our example, the aliasing relation $f = 9 * c/5 + 32$—be maintained as an invariant, which means that execution is aborted if the predicate becomes false.

A type declaration can also be viewed as an invariant, so it can be specified in a **constraints** clause. If we take the view that the type of a variable defines the set of values that variable can have, then declaring a variable f to be of type *real* is the same as requiring that the predicate $f \in \mathcal{R}$ be true throughout execution, where $\mathcal{R}$ is the set of real numbers.[1] Thus, we could eliminate the ": *real*" from the above **var** statement and add the constraint $x, y \in \mathcal{R}$. Since doing so would make the statement less readable, we will retain the customary syntax for type declarations.

Aliasing and typing can be viewed in terms of constraints because they are *static* properties. While *dynamic* properties, such as the values of variables, can be changed by execution of a program statement, static properties cannot. (In most languages, like the one considered here, a declaration is not a complete statement but rather part of a statement.) The methods we

[1] For simplicity, we assume $\mathcal{R}$ is the infinite set that mathematicians call the real numbers, thereby avoiding the problems that round-off errors would introduce for reasoning about equality of expressions.

develop for reasoning about aliasing and types can be used to reason about any static property.

Returning to aliasing, consider a more complicated example in which a program refers a point in terms of both its Cartesian coordinates x, y and its polar coordinates r, θ. Variables x, y, r, and θ are declared as follows.

$$
\begin{array}{l}
\textbf{var}\ \ x, y, r : \textit{real},\ \theta : [0, 2*\pi) \\
\quad \textbf{constraints}\ \ x = r*\cos(\theta)\ \ \textbf{and}\ \ y = r*\sin(\theta) \\
\qquad \textbf{in}\ S
\end{array}
$$

(The type declaration for θ states that it is a real in the range $0 \leq \theta < 2\pi$.) We would like this declaration to mean that when x is changed, r and θ are changed according to the constraints, but y is not. However, the fact that y should not change is based upon the knowledge that x and y are independent coordinates, which is not something discernible in the above statement. An additional constraint is needed to specify that assigning to x should not change the value of y and vice-versa; we write this constraint as $x \perp y$. Similarly, r and θ should be independent, so assigning a value to either r or θ does not change the other. Hence, the additional constraint $r \perp \theta$ is needed. The following declaration of x, y, r, and θ gives the desired aliasing relations.

$$
\begin{array}{l}
\textbf{var}\ \ x, y, r : \textit{real},\ \theta : [0, 2*\pi) \\
\quad \textbf{constraints}\ \ x = r*\cos(\theta)\ \textbf{and}\ y = r*\sin(\theta) \\
\qquad\qquad\qquad\quad \textbf{and}\ x \perp y\ \textbf{and}\ r \perp \theta \\
\qquad \textbf{in}\ S
\end{array}
$$

Finally, observe that the **var** statement can express forms of aliasing traditionally implemented by overlapping storage. The statement

$$
\begin{array}{l}
\textbf{var}\ \textit{full},\ \textit{right\_4} : \textit{natural} \\
\quad \textbf{constraints}\ \textit{right\_4} = \textit{full} \bmod 16 \\
\qquad \textbf{in}\ S
\end{array}
$$

aliases variable *right_4* to the right-most four bits of *full*, where *natural* denotes the nonnegative integers. Moreover, the declaration ensures the desired semantics even on a computer where integers are not stored in binary.

It is probably impossible for a compiler to handle our form of aliasing in all its generality. While the Fahrenheit/Celsius and *full*/*right_4* examples do not pose difficult compiling problems, consider what happens if the following statements appear in the body of the above **var** x, y, r, θ statement:

$\mathbf{read}(x,\ y)\ ;\ \mathbf{write}(\theta)$

Input values a, b with $a \neq 0$ produce the output value $\arctan(b/a)$—something no present-day compiler is likely to figure out.

We are interested in our general form of aliasing in order to reason about *implicit variables*—variables representing portions of the program state that are not directly visible to the programmer. For example, in a concurrent programming language with primitives to perform buffered message-passing, messages sent but not yet delivered are part of the state that must be described by implicit variables. (The ρ and σ multisets of [18] are such variables.) Implicit variables often involve complex aliasing relations. For some message-passing schemes, a channel is modelled by having an implicit variable in a sender aliased to an implicit variable in the receiver. Even more complex aliasing occurs when a channel emanating from a network is aliased to the union of the channels emanating from its components. The CSP language [10] supports such a hierarchical channel-naming scheme.

In the Generalized Hoare Logic (GHL) [12, 14], a logic for concurrent programs, one must reason about state components that describe the control state. In the original presentation of GHL, the control state was modelled by *at*, *in*, and *after* predicates, where $at(S)$ is true when control is at the entry point of statement S, $after(S)$ is true when control is at the exit point of statement S, and $in(S)$ is true when $at(S)$ is true or control is at a component of S. Axioms were given to describe the relations among these predicates. Thus, if S is the statement $S_1; S_2$, the axioms of GHL state:

$$\begin{aligned} at(S) &\equiv at(S_1) \\ after(S) &\equiv after(S_2) \\ in(S) &\equiv at(S) \lor in(S_1) \lor in(S_2) \\ after(S_1) &\equiv at(S_2) \end{aligned}$$

GHL included *ad hoc* rules for reasoning about these control predicates. However, by viewing the control predicates as implicit variables, and considering the above relations not as equality of predicates but as aliasing relations among variables, we can reason about the control state with exactly the same rules used to reason about the values of ordinary program variables. This is described in detail in [15].

2 Primitives for Constrained Execution

A **var** statement, like the one for the Cartesian/polar coordinate example, specifies three things:

- The names of new variables—x, y, r, and θ in the example.
- Constraints the new variables must satisfy, including those given explicitly by the **constraints** clause and those implicit in the type declarations. In the example, the constraints are:

$$\begin{array}{ll} x \in \mathcal{R} & x = r * \cos(\theta) \\ y \in \mathcal{R} & y = r * \sin(\theta) \\ r \in \mathcal{R} & x \perp y \\ \theta \in \mathcal{R} \ \wedge \ 0 \leq \theta < 2\pi & r \perp \theta \end{array}$$

- Other independence constraints involving the new variables. In the example, there is the implicit assumption that x, y, r, and θ are not aliased to any other variables, except perhaps variables declared in the body of the **var** statement. Thus, there are implicit constraints $x \perp q$ for all variables q declared outside the **var** statement, and similarly for y, r, and θ.

Instead of reasoning directly about the **var** statement, three primitive statements are introduced, each of which performs one of the above functions. These statements can be used to model the **var** statement and the aliasing of implicit variables described above.

The **new** statement is used to define new variable names, where

$$\textbf{new}\ x_1\ ,\ x_2\ ,\ \ldots\ ,\ x_n\ \textbf{in}\ S$$

defines x_1, x_2, ... , x_n to be new variable names for use within S. These variable names, plus any defined in a **new** statement containing this one, can be referenced from within S. The usual scoping rules apply, so that a variable x_i defined by this **new** statement is different from any other variables with the same name defined by a different **new** statement.

The **declare** statement is used to specify constraints. The statement

$$\textbf{declare}\ \ C\ \ \textbf{in}\ \ S$$

where C is a predicate, indicates that C is to be maintained during execution of S and that abortion is to occur if this becomes impossible. If

S contains a nondeterministic step, such as a nondeterministic assignment statement, then the choice must be made (if possible) so that the truth of C is maintained.

Finally, the **may alias** statement is used to specify independence relations implicit in a **var** statement. The statement

$$x \ \textbf{may alias} \ x_1, x_2, \ldots, x_n \ \textbf{in} \ S$$

specifies that, during execution of S, the value of x is independent of all variables, other than x_1, x_2, ... , x_n, declared in the context of this statement. Thus, this statement specifies constraints $x \perp q$ for all variables q not in the list x_1, x_2, ... , x_n. Although **may alias** specifies constraints, it cannot be modeled with a **declare** statement because that would require explicitly writing relations $x \perp p$ for every variable name p different from the x_i, and there could be an infinite number of such names.

The **new**, **declare**, and **may alias** statements can be used to model a **var** statement. The **var** x, y, r, θ example could be represented as:

$$
\begin{array}{l}
\textbf{new}\ x, y, r, \theta\ \textbf{in} \\
\quad x\ \textbf{may alias}\ x, y, r, \theta\ \textbf{in} \\
\quad\quad y\ \textbf{may alias}\ x, y, r, \theta\ \textbf{in} \\
\quad\quad\quad r\ \textbf{may alias}\ x, y, r, \theta\ \textbf{in} \\
\quad\quad\quad\quad \theta\ \textbf{may alias}\ x, y, r, \theta\ \textbf{in} \\
\quad\quad\quad\quad\quad\quad \textbf{declare}\ x \in \mathcal{R}\ \textbf{and}\ y \in \mathcal{R}\ \textbf{and} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad r \in \mathcal{R}\ \textbf{and}\ \theta \in [0, 2\pi)\ \textbf{and} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad x = r * \cos(\theta)\ \textbf{and}\ y = r * \sin(\theta) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \textbf{and}\ x \perp y\ \textbf{and}\ r \perp \theta \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \textbf{in}\ S
\end{array}
$$

In general, let x_1, ... , x_n, y_1, ... , y_m be $n+m$ distinct variable names, and let C be a predicate constructed from the m variables y_i plus zero or more of the x_i. The statement

$$\textbf{var} \ \ x_1 : T_1, \ldots, x_n : T_n \ \ \textbf{constraints} \ \ C \ \ \textbf{in} \ S$$

is modeled by

$$
\begin{array}{l}
\textbf{new}\ x_1, x_2, \ldots, x_n\ \textbf{in} \\
\quad x_1\ \textbf{may alias}\ x_1, \ldots, x_n, y_1, \ldots, y_m\ \textbf{in} \\
\quad\quad \ldots\ \ x_n\ \textbf{may alias}\ x_1, \ldots, x_n, y_1, \ldots, y_m\ \textbf{in} \\
\quad\quad\quad\quad\quad \textbf{declare}\ C\ \textbf{and}\ x_1 \in T_1\ \textbf{and} \\
\quad\quad\quad\quad \ldots\ \textbf{and}\ x_n \in T_n \\
\quad\quad\quad\quad\quad\quad\quad\quad \textbf{in}\ S
\end{array}
$$

3 Reasoning About Constraints

Our goal is to prove partial correctness formulas of the form $\{P\}\, S\, \{Q\}$, as first proposed by Hoare [8]. To reason about constrained execution, we interpret such a formula as a temporal assertion about the executions of S—namely, $\{P\}\, S\, \{Q\}$ is equivalent to the assertion that any terminating execution of S beginning with a state in which P is true ends in a state in which Q is true. Thus, we are viewing $\{P\}\, S\, \{Q\}$ as a temporal formula, which is not how it is usually viewed in Hoare's logic.

One reasons about such temporal assertions with temporal logic.[2] The only knowledge of temporal logic needed to understand this paper is that the temporal formula $\Box A$ is true of a statement S if and only if A remains true throughout every possible execution of S. The only formal rules for reasoning about temporal logic formulas that we need are the following, which are immediate consequences of the definition of $\Box$.

Strong Necessitation Rule:

$$\frac{P \Rightarrow Q}{\Box P \Rightarrow \Box Q}$$

Multiplicative Axiom:

$$\Box(A \wedge B) \;=\; \Box A \wedge \Box B$$

To apply temporal logic to program executions, we need to know what actions are atomic. For example, the formula $\Box A$ asserts that A is true before and after each atomic action. In general, an atomic action represents the execution of a primitive statement that can change the value of a variable. Execution of an assignment statement is the only kind of atomic action needed to describe the class of languages considered in this paper.

A program execution is a sequence of atomic actions. The possible executions of the statement

$$\textbf{declare}\ C\ \textbf{in}\ S$$

are all those executions of S for which C is true throughout the execution—that is, those executions for which $\Box C$ is true. This leads immediately to an inference rule for **declare**:

[2]See [16] for an elementary discussion of temporal logic and the appendix of [13] for the more advanced temporal logic needed to formalize our method.

declare *Rule*:

$$\frac{\Box C \Rightarrow \{P\}\ S\ \{Q\}}{\{P\}\ \textbf{declare}\ C\ \textbf{in}\ S\ \{Q\}}$$

In this rule, the hypothesis states that the predicate $\{P\}\,S\,\{Q\}$ is true or that C does not hold throughout all executions of S.

Note that in our temporal logic interpretation of partial correctness formulas, the pre- and postconditions are assertions about the program state, so they must be Boolean-valued functions on the state. In particular, they cannot contain temporal operators like $\Box$.

All the usual inference rules for partial correctness formulas (see [8]) still hold under this new interpretation. For example,

Rule of Consequence:

$$\frac{P' \vdash P,\ \{P\}\ S\ \{Q\},\ Q \vdash Q'}{\{P'\}\ S\ \{Q'\}}$$

This rule allows the precondition of a partial correctness formula to be weakened and the postcondition to be strengthened.

The only new general rule needed for reasoning about constraints is the

Constraint Strengthening Rule:

$$\frac{\{P \wedge C\}\ S\ \{Q \vee \neg C\}}{\Box C \quad \Rightarrow \quad \{P\}\ S\ \{Q\}}$$

To show the validity of this rule, observe that the hypothesis asserts that every terminating execution of S that begins with $P \wedge C$ true terminates with $Q \vee \neg C$ true. Another way of saying this is:

> For any terminating execution of S: C true of the initial state implies that if P is true of the initial state then Q is true of the final state or $\neg C$ is true of the final state.

The conclusion asserts the following:

> For any execution of S: C true throughout the execution implies that if P is true of the initial state and the execution terminates, then Q is true of the final state.

It should now be clear why the hypothesis implies the conclusion.

The expression $x \perp y$, introduced above for stating independence, can be given a precise meaning as a temporal formula. The formal definition is given in the Appendix. Intuitively, $x \perp y$ is the temporal formula asserting that if the next atomic action of the program is an assignment to x, then the assignment does not change the value of y, and if the next atomic action is an assignment to y, then it does not change the value of x.

The most general kind of temporal formula we write is of the form

$$\Box C \quad \Rightarrow \quad \{P\}\ S\ \{Q\}.$$

Because temporal formulas cannot appear in pre- and postconditions, $\perp$ relations can appear only in C, which means they appear only in the form $\Box(x \perp y)$. The temporal formula $\Box(x \perp y)$ asserts that no assignment to x during execution causes the value of y to change, and vice-versa.

Rules for reasoning about $\perp$ could be deduced from its formal definition. Instead, we state as axioms two properties that seem to be sufficient for reasoning about programs. An obvious axiom is:

> *Commutativity Axiom*:
>
> $$x \perp y \ = \ y \perp x$$

Another obvious axiom states that if x is always equal to y, then $\Box(y \perp z)$ implies $\Box(x \perp z)$. While this rule is sufficient for our examples, the following generalization is sometimes needed.

> *Substitution Axiom*: For any (single-valued) function f of n arguments and for any variables x, y_1, ... , y_n:
>
> $$\begin{array}{l} \Box\,(x = f(y_1, \ldots, y_n) \ \wedge \ y_1 \perp z \wedge \ldots \wedge y_n \perp z) \\ \quad \Rightarrow \quad \Box(x \perp z) \end{array}$$

4 Axioms For a Toy Language

In the preceding section, certain general rules for reasoning about temporal logic formulas were given. We now give language-specific rules and axioms for a toy language. The language contains the usual **skip**, assignment (:=), concatenation (;), and **while** constructs, in addition to the aforementioned **new**, **declare**, and **may alias** statements. As usual, there is one rule or axiom for each language construct. The rule for **declare** was already given.

4.1 Assignment

Consider the standard way of reasoning about assignment statements in Hoare's logic. Assuming there is no aliasing, one deduces

$$\{true\} \;\; x := y + 1 \;\; \{x = y + 1\}.$$

This formula is not valid when aliasing is allowed, because the assignment might appear in the body of a **var** statement that aliases x to equal y, in which case the postcondition $x = y + 1$ would be false. However, this postcondition would be satisfied if x and y were not aliased, which means that it would be satisfied if we constrained execution of $x := y + 1$ by requiring that $x \perp y$ hold. Hence, the rule to use when aliasing is possible is

$$\Box x \perp y \quad \Rightarrow \quad \{true\}\; x := y + 1 \;\{x = y + 1\}$$

(Remember that $x \perp y$ cannot appear in the precondition because temporal formulas may not appear in pre- or postconditions.)

More generally, the Assignment Axiom in Hoare's logic is

$$\{Q(exp, y_1, \ldots, y_n)\} \;\; x := exp \;\; \{Q(x, y_1, \ldots, y_n)\}$$

where $Q(x, y_1, \ldots, y_n)$ is any predicate involving only the program variables x, y_1, ... , y_n, and exp is an expression. Again, this axiom is valid only if x, the target of the assignment, is not aliased to any of the y_i's. Therefore, when aliasing is allowed, the correct formula is

Assignment Axiom:

$$\begin{array}{l} \Box(x \perp y_1 \;\wedge\; \ldots \;\wedge x \perp y_n) \quad \Rightarrow \\ \quad \{Q(exp, y_1, \ldots, y_n)\}\; x := exp \;\{Q(x, y_1, \ldots, y_n)\} \end{array}$$

4.2 May alias

Recall that the statement

$$x \;\; \textbf{may alias} \;\; x_1, x_2, \ldots, x_n \;\; \textbf{in} \;\; S$$

is really equivalent to

$$\textbf{declare}\; \mathcal{A} \;\textbf{in}\; S$$

where $\mathcal{A}$ is a conjunction of terms of the form $x \perp y$, for an infinite number of variables y—namely, every y that is not among the x_i. Therefore, the **declare** rule gives the following:

$$\frac{\Box\mathcal{A} \ \ \Rightarrow \ \ \{P\}\ S\ \{Q\}}{\{P\}\ x\ \textbf{may alias}\ x_1,\ \ldots,\ x_n\ \textbf{in}\ S\ \{Q\}} \tag{1}$$

Now, let $y_1,\ y_2,\ \ldots\ ,\ y_m$ be a finite number of variables, all different from any of the x_i. Then,

$$\mathcal{A} \ \Rightarrow \ (x \perp y_1 \ \wedge \ \ldots \ \wedge \ x \perp y_m). \tag{2}$$

Given

$$\Box(x \perp y_1 \ \wedge \ \ldots \ \wedge \ x \perp y_m) \ \ \Rightarrow \ \ \{P\}\ S\ \{Q\}$$

we use (2) to deduce

$$\Box\mathcal{A} \ \ \Rightarrow \ \ \{P\}\ S\ \{Q\}$$

which is the hypothesis of (1). Therefore, we get the following inference rule.

may alias *Rule*:

$$\frac{\begin{array}{l}\Box(x \perp y_1 \ \wedge \ \ldots \ \wedge \ x \perp y_m) \Rightarrow \\ \qquad \{P\}\ S\ \{Q\},\ \ \forall i,j : y_i \overset{\text{syn}}{\neq} x_j\end{array}}{\{P\}\ x\ \textbf{may alias}\ x_1,\ \ldots\ ,\ x_n\ \textbf{in}\ S\ \{Q\}}$$

where $a \overset{\text{syn}}{\neq} b$ means that a and b are syntactically different variable names.

4.3 New

The rule for the **new** statement is essentially the same as the one for ordinary Hoare triples—see Rule 16 of [1]. The statement

$$\textbf{new}\ \ x_1\ ,\ \ldots\ ,\ x_n\ \ \textbf{in}\ \ S$$

declares that the variables x_i are different from all variables declared outside the statement. It is equivalent to substituting for all free (undeclared) occurrences of x_i in S another variable y_i that is not used anywhere in the entire program. Of course, when reasoning about S in isolation, we do not know what the entire program is. However, since, we are concerned only with a particular pre- and postcondition, it suffices to choose the y_i so that they don't appear in that pre- or postcondition or in S. This leads to the following rule, where $S[y_1/x_1, \ldots, y_n/x_n]$ is the statement obtained by substituting y_i for every free occurrence of x_i in S, for $i = 1, \ldots, n$.

new *Rule*: For any distinct variable names $y_1, \ldots, y_n$ not occurring free in P, S, or Q:

$$\frac{\{P\}\ S[y_1/x_1, \ldots, y_n/x_n]\ \{Q\}}{\{P\}\ \mathbf{new}\ x_1\mathbf{,}\ \ldots\ \mathbf{,}\ x_n\ \mathbf{in}\ S\ \{Q\}}$$

Note that, unlike [1], initial values for the variables x_i in S are not assumed; executions containing arbitrary initial values are permitted.

4.4 Remaining Statements

The axioms for the remaining statements are just the ordinary Hoare logic partial correctness rules. For example, Hoare's rule for statement concatenation is:

Statement Concatenation:

$$\frac{\{P\}\ S\ \{Q'\}\,,\ \{Q'\}\ S'\ \{Q\}}{\{P\}\ S; S'\ \{Q\}}$$

5 Examples

5.1 No Aliasing

We first consider an example in which there is no aliasing—that is, there are no constraints. Let S be the statement

$$\begin{array}{l} \mathbf{var}\ \ x : \mathit{real}\ \ \mathbf{in}\ \ y := y + 1; \\ \phantom{\mathbf{var}\ \ x : \mathit{real}\ \ \mathbf{in}\ \ } x := y + 3 \end{array}$$

We will prove the obvious relation

$$\{y = 1\}\ S\ \{y = 2\} \tag{3}$$

From the Assignment Axiom, we get

$$\begin{array}{rcl} \Box \mathit{true} & \Rightarrow & \{y = 1\}\ y := y + 1\ \{y = 2\} \\ \Box x' \perp y & \Rightarrow & \{y = 2\}\ x' := y + 3\ \{y = 2\} \end{array}$$

Using ordinary propositional logic and the Strong Necessitation Rule, the antecedents of these implications can be strengthened to get

$$\Box(x' \perp y \wedge x' \in \mathcal{R}) \ \Rightarrow\ \{y = 1\}\ y := y + 1\ \{y = 2\} \tag{4}$$

$$\Box(x' \perp y \wedge x' \in \mathcal{R}) \ \Rightarrow\ \{y = 2\}\ x' := y + 3\ \{y = 2\} \tag{5}$$

Combining (4) and (5) gives

$$\begin{array}{l} \Box(x' \perp y \wedge x' \in \mathcal{R}) \;\Rightarrow \\ \{y = 1\}\ y := y + 1\ \{y = 2\}\ \wedge\ \{y = 2\}\ x' := y + 3\ \{y = 2\} \end{array}$$

Application of the Statement Concatenation Rule to the consequent of this yields

$$\begin{array}{l} \Box(x' \perp y \wedge x' \in \mathcal{R}) \;\Rightarrow \\ \qquad \{y = 1\}\ y := y + 1;\ x' := y + 3\ \{y = 2\} \end{array}$$

The **declare** Rule now allows us to conclude

$$\begin{array}{lll} \Box(x' \perp y) & \Rightarrow & \{y = 1\} \\ & & \textbf{declare}\ \ x' \in \mathcal{R}\ \textbf{in} \\ & & \quad y := y + 1;\ x' := y + 3 \quad \{y = 2\} \end{array}$$

Applying the **may alias** rule yields

$$\begin{array}{l} \{y = 1\} \quad x'\ \ \textbf{may alias}\ \ x', y\ \ \textbf{in} \\ \qquad\qquad \textbf{declare}\ \ x' \in \mathcal{R}\ \textbf{in} \\ \qquad\qquad\quad y := y + 1; x' := y + 3 \quad \{y = 2\} \end{array}$$

Finally, the **new** Rule allows us to deduce

$$\begin{array}{l} \{y = 1\} \quad \textbf{new}\ \ x\ \ \textbf{in} \\ \qquad\qquad x\ \ \textbf{may alias}\ \ x, y\ \ \textbf{in} \\ \qquad\qquad\quad \textbf{declare}\ \ x \in \mathcal{R} \\ \qquad\qquad\qquad \textbf{in}\ \ y := y + 1;\ x := y + 3 \quad \{y = 2\} \end{array}$$

The statement in this formula is equivalent to our original statement S, according to the method of modeling **var** statements described in Section 2, so the desired result is proved.

5.2 Simple Aliasing

Next, let S be the same as above except with x and y aliased:

$$\begin{array}{ll} \textbf{var}\ \ x : \mathit{real}\ \ \textbf{constraint}\ x = y\ \ \textbf{in} & y := y + 1; \\ & x := y + 3 \end{array}$$

Formula (3) is no longer valid for this program. Instead, we have

$$\{y = 1\}\ S\ \{y = 5\} \tag{6}$$

The proof of this is as follows. From the Assignment Axiom, we have

$$\begin{aligned} \Box \mathit{true} \quad &\Rightarrow \quad \{y = 1\}\ y := y + 1\ \{y = 2\} \\ \Box \mathit{true} \quad &\Rightarrow \quad \{y = 2\}\ x' := y + 3\ \{x' = 5\} \end{aligned}$$

Combining these, using the Statement Concatenation Rule, yields

$$\Box \mathit{true} \quad \Rightarrow \quad \{y = 1\}\ S'\ \{x' = 5\}$$

where S' is the statement

$$y := y + 1;\ x' := y + 3$$

Applying the Rule of Consequence to this, using the tautologies

$$\begin{aligned} &(x' = 5) \Rightarrow (y = 5\ \vee\ x' \neq y) \\ &(y = 1\ \wedge\ x' = y) \Rightarrow (y = 1) \end{aligned}$$

we conclude

$$\Box \mathit{true} \quad \Rightarrow \quad \{y = 1 \wedge x' = y\}\ S'\ \{y = 5 \vee x' \neq y\}$$

The Constraint Strengthening Rule now allows us to deduce

$$\Box x' = y \quad \Rightarrow \quad \{y = 1\}\ S'\ \{y = 5\}$$

Using propositional logic and the Strong Necessitation Rule, we can strengthen the antecedent of the implication to obtain

$$\Box(x' = y\ \wedge\ x' \in \mathcal{R}) \quad \Rightarrow \quad \{y = 1\}\ S'\ \{y = 5\}$$

The **declare** Rule now yields

$$\{y = 1\}\ \ \mathbf{declare}\ \ x' = y\ \mathbf{and}\ x' \in \mathcal{R}\ \ \mathbf{in}\ \ S'\ \ \{y = 5\}$$

from which the **may alias** Rule allows us to deduce

$$\begin{aligned} \{y = 1\}\quad & x'\ \ \mathbf{may\ alias}\ \ x', y\ \ \mathbf{in} \\ & \quad \mathbf{declare}\ \ x' = y\ \mathbf{and}\ x' \in \mathcal{R}\ \ \mathbf{in} \\ & \qquad S' \quad \{y = 5\} \end{aligned}$$

Finally, the **new** Rule allows the conclusion

$$\begin{aligned} \{y = 1\}\quad & \mathbf{new}\ \ x\ \ \mathbf{in} \\ & \quad x\ \ \mathbf{may\ alias}\ \ x, y\ \ \mathbf{in} \\ & \qquad \mathbf{declare}\ \ x = y\ \mathbf{and}\ x \in \mathcal{R}\ \ \mathbf{in} \\ & \qquad\quad y := y + 1 \\ & \qquad\quad x := y + 3 \quad \{y = 5\} \end{aligned}$$

This is just what is required to prove (6).

5.3 Cartesian/Polar Coordinates

As a final example, let S be the following statement.

$$\begin{array}{l} \textbf{var } x, y : \textbf{real} \\ \quad \textbf{constraints } x = r * \cos(\theta) \textbf{ and } y = r * \sin(\theta) \\ \qquad\qquad\qquad \textbf{and } x \perp y \ \ \textbf{in } x := 2 * x; \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad y := 2 * y \end{array}$$

A little trigonometry shows that if r is initially positive, then executing S should double the value of r and leave θ unchanged. Thus, the following should hold:

$$\{r = r_0 \ \wedge \ \theta = \theta_0\} \ \ S \ \ \{r = 2r_0 \ \wedge \ \theta = \theta_0\}$$

However, further reflection indicates that this is not quite valid because executing S can add any even multiple of π to θ or can negate r and add any odd multiple of π to θ. Thus, we stipulate that $r \geq 0$ and $0 \leq \theta < 2\pi$ remain true throughout execution, and prove

$$\begin{array}{l} \Box(r \geq 0 \ \wedge \ 0 \leq \theta < 2\pi) \ \Rightarrow \\ \quad \{r = r_0 \wedge \theta = \theta_0\} \ \ S \ \ \{r = 2r_0 \ \wedge \ \theta = \theta_0\} \end{array} \tag{7}$$

Let S' be the statement

$$x' := 2 * x' \,; \, y' := 2 * y'.$$

From the Statement Concatenation Rule and two applications of the Assignment Axiom, we deduce

$$\begin{array}{l} \Box x' \perp y' \ \ \Rightarrow \ \ \{x' = r_0 \cos\theta_0 \ \wedge \ y' = r_0 \sin\theta_0\} \ \ S' \\ \qquad\qquad\qquad\qquad\qquad \{x' = 2r_0 \cos\theta_0 \ \wedge \ y' = 2r_0 \sin\theta_0\} \end{array}$$

Now, note that the following are tautologies:

$$\begin{array}{l} (x' = r_0 \cos\theta_0 \ \wedge \ r\cos\theta = r_0 \cos\theta_0 \ \wedge \ x' = r\cos\theta) \\ \qquad \Rightarrow (x' = r_0 \cos\theta_0) \end{array}$$

$$(x' = 2r_0 \cos\theta_0) \Rightarrow (r\cos\theta = 2r_0 \cos\theta_0 \ \vee \ x' \neq r\cos\theta)$$

Similar tautologies apply to y'. Therefore, by the Rule of Consequence we conclude

$$\begin{array}{l}
\Box x' \perp y' \quad \Rightarrow \\
\quad \left\{ \begin{array}{l} x' = r_0 \cos\theta_0 \wedge r\cos\theta = r_0\cos\theta_0 \wedge x' = r\cos\theta \;\; \wedge \\ y' = r_0 \sin\theta_0 \;\wedge r\sin\theta = r_0\sin\theta_0 \;\wedge y' = r\sin\theta \end{array} \right\} \\
\qquad\qquad\qquad S' \\
\quad \left\{ \begin{array}{l} (r\cos\theta = 2r_0\cos\theta_0 \vee x' \neq r\cos\theta) \wedge \\ (r\sin\theta = 2r_0\sin\theta_0 \;\vee y' \neq r\sin\theta) \end{array} \right\}
\end{array}$$

From this, the Constraint Strengthening Rule and Rule of Consequence allow us to derive

$$\begin{array}{l}
\Box(x' \perp y' \wedge\; x' = r\cos\theta \;\wedge\; y' = r\sin\theta) \quad \Rightarrow \\
\quad \left\{ \begin{array}{l} r\cos\theta = r_0\cos\theta_0 \wedge \\ r\sin\theta = r_0\sin\theta_0 \end{array} \right\} S' \left\{ \begin{array}{l} r\cos\theta = 2r_0\cos\theta_0 \wedge \\ r\sin\theta = 2r_0\sin\theta_0 \end{array} \right\}
\end{array}$$

By the Rule of Consequence, using some trigonometry and the previous theorem we can now deduce

$$\begin{array}{l}
\Box(x' \perp y' \wedge x' = r\cos\theta \;\wedge\; y' = r\sin\theta) \quad \Rightarrow \\
\left\{ \begin{array}{l} r = r_0 \wedge \theta = \theta_0 \wedge \\ r \geq 0 \;\; \wedge 0 \leq \theta < 2\pi \end{array} \right\} S' \left\{ \begin{array}{c} (r = 2r_0 \wedge \theta = \theta_0) \vee \\ \neg(r \geq 0 \quad \wedge 0 \leq \theta < 2\pi) \end{array} \right\}
\end{array}$$

We now use the Constraint Strengthening Rule to derive

$$\begin{array}{l}
\Box(x' \perp y' \wedge x' = r\cos\theta \;\wedge\; y' = r\sin\theta \wedge \\
\qquad r \geq 0 \;\wedge\; 0 \leq \theta < 2\pi) \quad \Rightarrow \\
\qquad\qquad \{r = r_0 \;\wedge\; \theta = \theta_0\} \;\; S' \;\; \{r = 2r_0 \;\wedge\; \theta = \theta_0\}
\end{array}$$

It is now a simple matter to use the **declare** Rule, the **may alias** Rule, and finally the **new** Rule to obtain (7).

6 Discussion

We have introduced the idea of describing types and aliasing in terms of constraints and given general rules for reasoning about constrained execution. Our approach involves embedding the usual Hoare partial correctness formalism in temporal logic. One reasons about static properties with constraints and about dynamic properties with pre- and postconditions.

Having applied our method to a simple language, we now consider some of the problems in extending it to more complex languages. We also discuss the relation of our approach to previous work.

6.1 Types

In our toy language, we were able to handle a type declaration simply by translating it to a constraint about the values that the variable can assume. This does not work for languages that make more extensive use of type information—for example, by performing coercions in the event of a type mismatch; nor does it work for languages in which a type mismatch in an assignment generates an indeterminate result rather than abortion. (Our semantics causes abortion if executing an assignment would violate a type constraint.)

Reasoning about these more complex languages requires adding state predicates that characterize the type of a variable and modifying our Axiom of Assignment. However, care must be employed when reasoning about predicates like $\mathbf{type}(x) = $ '*integer*' because $x = y$, which means that the values of x and y are equal, does not imply $\mathbf{type}(x) = \mathbf{type}(y)$.

If a type mismatch can abort execution, reasoning about type correctness requires proving total correctness properties. While we have not yet considered this problem, we feel that our approach should be ideal for proving termination properties because it is based upon temporal logic, and temporal logic is effective for proving liveness properties like termination [16].

6.2 Generalized Assignments: Expressions as Targets

In the Cartesian/polar coordinate example, x is aliased to $r \cos\theta$. Thus, assigning to x is the same as assigning to $r \cos\theta$. The obvious next step is to try writing $r \cos\theta$ on the left-hand side of an assignment statement, even if there is no variable aliased to this expression. For arbitrary expressions exp_1 and exp_2, the statement

$$exp_1 \ := \ exp_2$$

causes the value of exp_1 after execution to be the same as the value of exp_2 before execution. To reason about this form of generalized assignment, we extend $\perp$ to be a relation on expressions rather than just on variable names. The temporal formula $exp_1 \perp exp_2$ now means that assigning to exp_1 does not change the value of exp_2, and vice-versa. The axiom for generalized assignment is the same as the Assignment Axiom given above, except that x and the y_i can be arbitrary expressions. The commutativity and substitution axioms given above are also valid for these more general $\perp$

relations. However, additional axioms are needed for deriving $\perp$ relations between expressions from $\perp$ relations between their components—axioms such as

$$(exp_1 \perp exp_2) \wedge (exp_1 \perp exp_3) \;\Rightarrow\; exp_1 \perp (exp_2 + exp_3)$$

We do not give a formal semantics for this here.

6.3 Arrays and Pointers

Our approach can handle arrays by regarding assignment to an element of an array as an assignment to the entire array, as described in [11]. Array assignment cannot be handled using our generalized assignment statement, where an expression like $A[i]$ appears on the left-hand side, because this does not give the usual semantics for

$$A[exp] \;:=\; exp'.$$

Letting the subscripts *old* and *new* denote values before and after the assignment, the semantics of generalized assignment defines the above statement to mean that $A_{new}[exp_{new}] = exp'_{old}$, while the usual meaning of array assignment is $A_{new}[exp_{old}] = exp'_{old}$. This difference helps explain why the ordinary assignment axiom is not easily extended to arrays. This also indicates why our more general assignment statement is not easily compiled, since it requires computing the value of an expression in a (new) state without knowing what that state is.

By regarding an array as a single variable, our formalism can handle aliasing relations between entire arrays. However, our current formalism does not handle simple aliasing of array elements. For example, if x is not aliased to any element of the array A, then we can easily prove that assigning a value to $A[1]$ does not change x's value by deducing

$$\Box x \perp A \quad\Rightarrow\quad \{x = 7\}\; A[1] \;:=\; 1\; \{x = 7\}$$

However, we cannot do this just knowing that x is not aliased to $A[1]$, because our rules are not strong enough to prove

$$\Box x \perp A[1] \quad\Rightarrow\quad \{x = 7\}\; A[1] \;:=\; 1\; \{x = 7\}$$

The required generalization must replace $\perp$ with a noncommutative relation, since the assertion that assigning to $A[i]$ does not change i is not equivalent to the assertion that assigning to i does not change $A[i]$. Moreover, aliasing relations are no longer static, since whether $A[i]$ and x are aliased may depend upon the value of i. Being dynamic, aliasing relations like $\perp$ have to appear in pre- and postconditions, which is prohibited by the current formalism.

Similar extensions are needed for reasoning about aliasing in programs that use pointers. Moreover, in a language with pointers, type relations might also be dynamic—for example, if a pointer can point to variables of type *real* and of type $[0, 2 * \pi)$. In this case, type relations would have to appear in pre- and postconditions.

These extensions will be described in a future paper.

6.4 Procedures

The most general form of parameter passing is call by name, since it can be used to simulate call by reference and call by value-result. With call by name, a formal parameter is essentially aliased to the corresponding argument. Thus, our approach can be used for reasoning about procedures.

Traditionally, programming languages with procedures do not allow an arbitrary expression as the argument corresponding to a formal parameter that appears on the left-hand side of an assignment, since assignment to an expression is not defined by these languages. We have defined what it means to assign a value to a variable that is aliased to an expression, so there is no semantic reason for this prohibition. However, some restriction is needed to ensure that the language can be compiled.

6.5 Related Work

Previous work on aliasing, [1, 3, 4, 5, 6, 7, 9], has been motivated by shared storage among arguments of a procedure call. We are aware of no work that can handle the rich aliasing structures that concern us. However, programming languages where computations are partially or completely specified in terms of constraints have been investigated [2, 19, 20, 21, 22].

In most previous work on aliasing, the program state consists of a mapping from variable names to a space of (abstract) locations plus a map from locations to values. We feel that if the language itself has no pointers, then the semantics should not be given in terms of pointers, even if the values of these pointers are abstract locations instead of real memory addresses. Moreover, the existence of semantic pointers unnecessarily complicates reasoning about programs. Also, approaches based on locations are rarely fully abstract [3]. Finally, and most importantly, the use of locations does not support reasoning about the more general form of aliasing that is not based on shared storage.

Our work resembles Reynolds' [17] handling of call by name in many ways. Reynolds defines a formal system, called Specification Logic, that contains a relation $\#$ very similar to $\perp$ and an assignment rule much like ours. The effects of aliasing are described in terms of *interference*, which can be seen as the dual of our viewing aliasing in terms of *invariants*. (See [14] for a discussion of the relation between interference and invariance.)

The meaning of a formula in Specification Logic is based on an *environment* in addition to a state. The environment is a mapping from variable names to a space of locations and brings with it the difficulties mentioned above. In addition, in Specification Logic, assertions about the environment are made using a completely new logic, distinct from the one used to reason about the state. In our approach, all reasoning is done in a single logical system—temporal logic. Assertions about the state are expressed by temporally trivial formulas—formulas containing no temporal operators. We do not need the concept of an environment, using instead temporal assertions about the state. Of course, Specification Logic handles forms of aliasing not considered in this paper. We are currently extending our temporal logic approach to cover the full range of language features handled by Specification Logic.

Our approach can be viewed as a generalization of one proposed by Brooks [3], and we were somewhat influenced by his work. We handle a much more general form of aliasing, but, if we restricted ourselves to aliasing relations that are simple equalities between variables, then proofs in our system and in Brooks' would be quite similar. To extend Brooks' work to handle our more general form of aliasing, it appears that a new deductive method would have to be added; we avoided this by embedding our proof system in temporal logic.

Acknowledgments

We would like to thank A. Demers, C. Cartwright, W. P. De Roever, D. Gries, J. Hook, A. Meyer, and the members of IFIP Working Groups 2.2 and 2.3 for helpful discussions.

References

[1] K. R. Apt. Ten Years of Hoare's Logic: A Survey. *Trans. on Programming Languages and Systems 3*, 4 (October 1981), 431-483.

[2] Alan Borning. The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. *Trans. on Programming Languages and Systems 3*,4 (Oct. 1981), 353-387.

[3] Stephen D. Brooks. A Fully Abstract Semantics And A Proof System For An ALGOL-like Language With Sharing. Preliminary Draft, Carnegie-Mellon University, Feb. 1984.

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

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

[6] J. Halpern, A. Meyer, and B. Trakhtenbrot. From Denotational to Operational and Axiomatic Semantics for ALGOL-like Languages: An Overview. *Proc. 1983 Workshop on Logics of Programs*, Lecture Notes in Computer Science, Volume 164, Springer-Verlag, 1984.

[7] D. Gries and G. Levin. Assignment and Procedure Call Proof Rules. *Trans. on Programming Languages and Systems 2*, 4 (Oct. 1980), 564-579.

[8] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM 12*, 10 (October 1969), 576-580.

[9] C. A. R. Hoare. Procedures and Parameters: An Axiomatic Approach. *Symposium on Semantics of Algorithmic Languages*, Lecture Notes in Computer Science, Volume 88, Springer-Verlag, New York, 1971.

[10] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM 21*, 8 (August 1978), 666-677.

[11] C. A. R. Hoare and N. Wirth. An Axiomatic definition of the programming language Pascal. *Acta Informatica 2* (1973), 335-355.

[12] L. Lamport. The "Hoare Logic" of Concurrent Programs. *Acta Informatica 14* (1980), 21-37.

[13] L. Lamport. Specifying Concurrent Program Modules. *Trans. on Programming Languages and Systems 5*, 2 (April 1983), 190-222.

[14] L. Lamport and F. B. Schneider. The "Hoare Logic" of CSP and All That. *Trans. on Programming Languages and Systems 6*, 2 (April 1984), 281-296.

[15] L. Lamport and F. B. Schneider. Fully Compositional Generalized Hoare Logic. In preparation.

[16] S. S. Owicki and L. Lamport. Proving Liveness Properties of Temporal Logic. *Trans. on Programming Languages and Systems 4*, 3 (July 1982), 455-495.

[17] J. Reynolds. *The Craft of Programming.* Prentice Hall International, London, 1981.

[18] R. D. Schlicting and F. B. Schneider. Using Message Passing for Distributed Programming: Proof Rules and Disciplines. *Trans. on Programming Languages and Systems 6*, 3 (July 1984), 402-431.

[19] R. M. Stallman and G. J. Sussman. Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis. *Ariticial Intelligence 9*, 1977, 135-196.

[20] G. L. Steele Jr., and G. J. Sussman. Constraints. *Proceedings APL'79, ACM SIGPLAN STAPL APL Quote Quad 9*, 4 (June 1979), 208-225.

[21] G. J. Sussman, and R. M. Stallman. Heuristic Techniques in Computer-Aided Circuit Analysis. *IEEE Transactions on Circuits and Systems CAS-22*, 11 (November 1975).

[22] I. E. Sutherland. SKETCHPAD: A Man-Machine Graphical Communication System. M.I.T. Lincoln Laboratory Technical Report 296, (January 1963).

Appendix
The Formal Semantics

A.1 The Language

We now give a formal semantics for our toy language containing **skip**, assignment, concatenation, and **while**, plus the three statements **new**, **declare**, and **may alias** introduced to model the **var** statement. The class of expressions and variable types is not specified. We assume only that expressions are built from some set *Var* of variable names, that variables assume values in some set *Val*, and that expressions are built from operators on those values. However, in the statement

$$\textbf{declare } C \textbf{ in } S$$

C can involve $\perp$ in addition to Boolean expressions.

Finally, we require the value of an expression to be defined for any values of its component variables. Thus, the expression $x + 10$ must be assigned a value, even when $x = \mathit{true}$. This can be done by including a special value *undefined* in *Val*; the precise details for doing this are irrelevant.

A.2 Temporal Logic

A *state* is defined to be a mapping from *Var* to *Val*, so a state s assigns a value $s(x)$ to every variable name $x \in \mathit{Var}$. Let $\mathcal{S}$ denote the set of states. A state s is extended to a mapping from ordinary (nontemporal) expressions to values in the obvious way—for example, $s(x + y)$ is defined to equal $s(x) + s(y)$. Let $s \models \mathit{exp}$ denote the assertion that $s(\mathit{exp}) = \mathit{true}$. ($s \models x > 10$ is false if $s(x)$ has a nonnumeric value.)

An *action* is defined to be an element of $\mathit{Var} \cup \{\tau\}$, where τ is a symbol not in *Var*. For $x \in \mathit{Var}$, action x represents an assignment to x. Action τ represents an assignment to a variable declared in some **new** statement inside the current statement; thus τ, models a variable that is "invisible" in the current context.

A *behavior* is defined to be a sequence σ of the form

$$s_0 \xrightarrow{x_1} \cdots \xrightarrow{x_n} s_n \tag{8}$$

where the s_i are states and the x_i are actions. This behavior denotes an execution starting in state s_0 and terminating in state s_n, where the i^{th} action changes the state from s_{i-1} to s_i. Since partial correctness does not distinguish between aborting and infinite looping, we consider only finite (terminating) behaviors, although our definitions are easily extended to include infinite (nonterminating) ones. We allow the case $n = 0$, where s_0 is the behavior starting in state s_0 that performs no actions.

In the temporal logic of [13], a formula is composed of state predicates, action predicates, and temporal operators. The set of Boolean expressions is taken to be our state predicates. Action predicates are those of the form $\alpha(x)$, together with the action predicate *halt*. A Boolean expression is true of a behavior if it is true of the first state in the behavior. An action predicate $\alpha(x)$ is true if x is the first action in the behavior; action predicate *halt* is true only if there is no next action. The semantics of this temporal logic assigns a truth value to $\sigma \models F$ for every behavior σ and every formula F. We write $\models F$ to denote that $\sigma \models F$ is true for all behaviors σ.

We will define a behavioral semantics for our language where $\mathcal{M}[\![S]\!]$ is a set of behaviors representing all possible terminating executions of S. Semantic validity of a temporal logic formula F for a program S is defined by

$$\models_S F \stackrel{\text{def}}{=} \forall \sigma \in \mathcal{M}[\![S]\!] : \sigma \models F$$

To define the temporal operator $\perp$, we first define the operator $\lrcorner$ by letting $\sigma \models x \lrcorner y$ mean that if the first action of σ is an assignment to x, then that assignment does not change the value of y. In other words, letting σ be the sequence of (8), we have

$$\sigma \models x \lrcorner y \stackrel{\text{def}}{=} (x \stackrel{\text{syn}}{=} x_1) \Rightarrow (s_0(y) = s_1(y))$$

In the temporal logic of [13],

$$x \lrcorner y \stackrel{\text{def}}{=} \forall \eta : (y = \eta) \Rightarrow (\alpha(x) \lhd y = \eta)$$

We define $x \perp y$ to be $(x \lrcorner y) \wedge (y \lrcorner x)$. Note that $\sigma \models x \perp y$ is true if σ is a sequence with no actions.

The formulas deduced by our method for reasoning about programs are of the form $\Box C \Rightarrow \{P\}\, S\, \{Q\}$, where C is a temporal logic formula. However, $\{P\}\, S\, \{Q\}$ is not a temporal logic formula because it refers to the statement S—a concept with no counterpart in the temporal logic. To make semantic sense out of this formula, first define $\{P\} \rightarrow \{Q\}$ to be true for the behavior (8) if and only if $s_0 \models P \Rightarrow s_n \models Q$. This is defined in terms of temporal logic operators by

$$\{P\} \rightarrow \{Q\} \stackrel{\text{def}}{=} P \Rightarrow \Box(halt \Rightarrow Q)$$

A program S satisfies $\Box C \Rightarrow \{P\}\, S\, \{Q\}$ if and only if $\models_S \Box C \Rightarrow \{P\} \rightarrow \{Q\}$.

A.3 The Behavioral Semantics

For any statement S in our language, we define $\mathcal{M}[\![S]\!]$ to be a set of behaviors. The definition is by induction on the structure of S.

skip $\mathcal{M}[\![\textbf{skip}]\!] \stackrel{\text{def}}{=} \{s_0 : s_0 \in \mathcal{S}\}$

The **skip** statement generates no actions.

assignment $\mathcal{M}[\![x := exp]\!] \stackrel{\text{def}}{=} \{s \xrightarrow{x} t : t(x) = s(exp)\}$

An assignment generates a single action that sets the value of the left-hand side to the original value of the right-hand side. Note that $\mathcal{M}[\![S]\!]$ contains behaviors that make arbitrary changes to other variables, since any such change could be caused by an appropriate aliasing.

concatenation $\mathcal{M}[\![S_1;\ S_2]\!]$ is defined to equal

$$\begin{array}{l} \{s_0 \xrightarrow{x_1} \cdots \xrightarrow{x_{n+m}} s_{n+m} : \\ \quad s_0 \xrightarrow{x_1} \cdots \xrightarrow{x_n} s_n \in \mathcal{M}[\![S_1]\!] \\ \quad \text{and } s_n \xrightarrow{x_{n+1}} \cdots \xrightarrow{x_{n+m}} s_{n+m} \in \mathcal{M}[\![S_2]\!]\} \end{array}$$

Note that we are including only finite (terminating) behaviors.

while We define $\mathcal{M}[\![\textbf{while}\ B\ \textbf{do}\ S]\!]$ inductively by

$$\begin{array}{lcl} \mathcal{M}[\![\textbf{while}_0\ B\ \textbf{do}\ S]\!] & \stackrel{\text{def}}{=} & \{s_0 \in \mathcal{S} : s_0(B) = \mathit{false}\} \\ \mathcal{M}[\![\textbf{while}_{i+1}\ B\ \textbf{do}\ S]\!] & \stackrel{\text{def}}{=} & \\ \quad \{s_0 \xrightarrow{x_1} \cdots \xrightarrow{x_n} s_n \in\ \mathcal{M}[\![S;\ \textbf{while}_i\ B\ \textbf{do}\ S]\!] : & & \\ \quad\quad s_0(B) = \mathit{true}\} & & \\ \mathcal{M}[\![\textbf{while}\ B\ \textbf{do}\ S]\!] & \stackrel{\text{def}}{=} & \bigcup_{i=0}^{\infty} \mathcal{M}[\![\textbf{while}_i\ B\ \textbf{do}\ S]\!] \end{array}$$

Intuitively, $\mathcal{M}[\![\textbf{while}_i \ldots]\!]$ contains the behaviors in which the body of the **while** statement is executed exactly i times.

declare $\mathcal{M}[\![\textbf{declare}\ C\ \textbf{in}\ S]\!] \stackrel{\text{def}}{=} \{\sigma \in \mathcal{M}[\![S]\!] : \sigma \models \Box C\}$

The **declare** acts as a "filter" to eliminate any behaviors of S in which C does not always hold.

new If σ is the sequence (8), let $\sigma[x_1/y_1, \ldots, x_n/y_n]$ denote the sequence

$$s'_0 \xrightarrow{x'_1} \cdots \xrightarrow{x'_n} s'_n$$

where

$$s'_i(v) \quad = \quad \begin{cases} s_i(v) & \text{if } \forall j : v \stackrel{\text{syn}}{\neq} x_j \text{ and } v \stackrel{\text{syn}}{\neq} y_j \\ s_0(x_j) & \text{if } v \stackrel{\text{syn}}{=} x_j \\ s_0(y_j) & \text{if } v \stackrel{\text{syn}}{=} y_j \end{cases} \tag{9}$$

$$x'_i \quad = \quad \begin{cases} x_i & \text{if } \forall j : x'_i \overset{\text{syn}}{\neq} y_j \\ \tau & \text{otherwise} \end{cases} \tag{10}$$

Then $\sigma \in \mathcal{M}[\![\textbf{new } x_1, \ldots, x_n \textbf{ in } S]\!]$ if and only if there exist variable names $y_1, \ldots, y_n$ not free in S and $\sigma' \in \mathcal{M}[\![S[y_1/x_1, \ldots, y_n/x_n]]\!]$ such that $\sigma = \sigma'[x_1/y_1, \ldots, x_n, y_n]$.

This formal definition captures the intuitive notion that to execute the **new** statement, one first executes its body with new variables y_i substituted for the x_i. The resulting execution is then modified by hiding all references to the variables y_i—replacing assignments to the y_i by τ actions and letting y_i refer once more to its external declaration—and requiring that the externally declared values of the x_i remain unchanged.

may alias $\mathcal{M}[\![x \textbf{ may alias } x_1, \ldots, x_n \textbf{ in } S]\!]$ is defined to equal

$$\{\sigma \in \mathcal{M}[\![S]\!] : \forall y : (\forall i : y \overset{\text{syn}}{\neq} x_i) \ \Rightarrow \ \sigma \models \Box(x \perp y)\}$$

This is the formal definition of our intuitive idea that the **may alias** is equivalent to a declaration of an an infinite number of $\perp$ relations.

A.4 Soundness and Completeness

In the main body of this paper, we gave a set of rules for deriving formulas of the form $\Box C \ \Rightarrow \ \{P\}\, S\, \{Q\}$. Having defined the set of behaviors $\mathcal{M}[\![S]\!]$, we have given a semantic meaning to these formulas, namely:

$$\mathcal{M}[\![\Box C \ \Rightarrow \ \{P\}\, S\, \{Q\}]\!] \ \stackrel{\text{def}}{=} \ \models_S \Box C \Rightarrow \{P\} \ \rightarrow \ \{Q\}$$

We can now discuss the soundness and completeness of our system.

Soundness means that for any formula F derived by our system, $\mathcal{M}[\![F]\!]$ equals *true*. The proof of this involves checking the validity of all our axioms and proof rules. This involves a straightforward formalization of the informal arguments given in section 4.

Completeness means that every semantically correct formula is derivable using our rules. Since completeness is impossible, one usually proves relative completeness in the sense of Cook, which, as explained in [1], means that the system is complete if we assume that:

C1. All valid state predicates are given.

C2. The set of state predicates is sufficiently expressive, meaning that for any state predicate P and statement S, $post_S(P)$, the strongest postcondition of P with respect to S, is a state predicate.

These assumptions are not enough to guarantee completeness in our system. In ordinary Hoare logic, one assumes the ability to reason about state predicates. Since formulas in our logic contain temporal operators, like $\Box$ and $\perp$, we need to reason about temporal logic formulas. We therefore assume that[3]

C1′. All valid temporal logic formulas constructed from the state predicates are given.

where a *valid temporal logic formula* is one that is true for any behavior. Thus, just as assumption C1 for ordinary Hoare Logic contains only information about the state space—not information about the program—so C1′ gives information about state functions and temporal operators—not about the program. Since state predicates are temporal formulas, C1′ subsumes C1.

In addition to strengthening C1, we must also strengthen C2. To see why, suppose our set of state predicates did not contain the predicate *true*, but required that we use the semantically equivalent predicate $x = x$. Our Assignment Axiom does not allow us to deduce $\{x = x\}\, y := 1\, \{x = x\}$; we can only deduce it under the irrelevant hypothesis $\Box(y \perp x)$. In general, we need to assume that if a state predicate does not depend upon the value of a variable x, then we can write that predicate as an expression that does not involve x. We therefore require the following additional expressiveness condition:

C2a. For any expression exp, mathematical function f, and variables y_1, … , y_n: if the relation $exp = f(y_1, \ldots, y_n)$ is valid, then $f(y_1, \ldots, y_n)$ is an expression.

Having made these extra assumptions, we now consider completeness. Completeness for ordinary Hoare logic means that every valid formula is provable. In our system, there are formulas of the form $\Box C \Rightarrow \{P\}\, S\, \{Q\}$, where C has the form $C' \wedge (x \perp y_1) \wedge \ldots \wedge (x \perp y_n)$ and C' is an ordinary (nontemporal) expression. Completeness therefore means that every valid formula of this form is provable.

[3] Actually, this assumption is stronger than necessary, since we are concerned only with temporal logic formulas of the form $\Box C \Rightarrow \{P\}\, S\, \{Q\}$.

The proof is by induction on the structure of S. If S is a **skip** statement, then $\Box C \;\Rightarrow\; \{P\}\,S\,\{Q\}$ is semantically equivalent to $(C' \wedge P) \Rightarrow Q$. By assumption C1$'$, this is provable, and we can use it, the ordinary axiom for **skip** ($\{P\}\,\mathbf{skip}\,\{\mathbf{P}\}$), the Rule of Consequence, and the Constraint Strengthening Rule to deduce $\Box C \;\Rightarrow\; \{P\}\,S\,\{Q\}$. (The details are left to the reader.)

Next, let S be the assignment statement $x := \mathit{exp}$. A consequence of C1$'$ is a complete system for deriving $\perp$ relations. Thus, we can assume that C includes all relations $x_i \perp y_i$ that are derivable from it. It is easy to see that $\Box C \;\Rightarrow\; \{P\}\,S\,\{Q\}$ is semantically equivalent to, and, by the Constraint Strengthening Rule, derivable from

$$\Box(x_1 \perp y_1 \,\wedge\, \ldots \,\wedge\, x_n \perp y_n) \quad\Rightarrow\quad \{P \wedge C'\}\; S\; \{Q \vee \neg C'\}$$

We can therefore restrict consideration to the case in which C consists only of the conjunction of the constraints $x_i \perp y_i$.

Since $\{P\}\,S\,\{Q\}$ is semantically equivalent to $\mathit{post}_S(P) \;\Rightarrow\; Q$, by the Rule of Consequence and C2, we can let $Q = \mathit{post}_S(P)$.

Let $Q = Q(x, z_1, \ldots, z_m)$, where the z_i are different from x. By C2a, we can assume that the value of Q depends upon each of the z_j. Since $Q = \mathit{post}_S(P)$, this means that there is a behavior $s \xrightarrow{x} t$ in $\mathcal{M}[\![S]\!]$ such that $s \models P$, but $t' \not\models Q$ for some state t' that differs from t only in the value of z_j. However, since the semantics of S does not constrain the ending values of any variable other than x, $s \xrightarrow{x} t'$ is in $\mathcal{M}[\![S]\!]$. Therefore, if $x \perp z_j$ were not one of the constraints in C, then $\Box C \;\Rightarrow\; \{P\}\,S\,\{Q\}$ would be invalid. Hence, the constraint C contains all the $\perp$ relations needed to apply our Assignment Axiom, and completeness follows by the same argument as in the ordinary Hoare Logic [1].

The proof for the **declare** is immediate, since if S is the statement

$$\textbf{declare}\;\; C' \;\;\textbf{in}\;\; S'$$

then $\Box C \;\Rightarrow\; \{P\}\,S\,\{Q\}$ is semantically equivalent to $\Box(C \wedge C') \Rightarrow \{P\}\,S'\,\{Q\}$. Similarly, if S is the statement

$$x \;\textbf{may alias}\; x_1, \ldots\, x_n \;\textbf{in}\; S'$$

the result follows from the fact that $\Box C \;\Rightarrow\; \{P\}\,S\,\{Q\}$ is semantically equivalent to $\Box C \wedge C' \;\Rightarrow\; \{P\}\,S'\,\{Q\}$, where $C' \;\equiv\; (x \perp y_1) \,\wedge\, \ldots \,\wedge\, (x \perp y_m)$ and the y_i are all the variable names other than the x_j that appear in P, C, and Q.

To finish the completeness proof, we must show that for every compound statement S, we can prove every valid formula $\Box C \Rightarrow \{P\}\, S\, \{Q\}$ under the assumption that we can prove every such formula for the components of S. This involves a separate proof for every type of compound statement. The proofs for concatenation, **while**, and **new** are similar to the ones for the ordinary Hoare logic given in [1]. The only difference in the proofs arises because of the "$\Box C \Rightarrow$". The proofs in [1] rely on the fact that $\{P\}\, S\, \{Q\}$ is valid iff $post_S(P) \Rightarrow Q$. The formula $\Box C \Rightarrow \{P\}\, S\, \{Q\}$ is valid iff $\{P\}$ **declare** C **in** $S\, \{Q\}$ is valid, by the semantic equivalence mentioned above. This, in turn, is valid iff $post_{\mathbf{declare}\ C\ \mathbf{in}\ S}(P)$ is expressable, and implies Q. However, expressability follows from our assumptions C2 and C2a. With this observation, the completeness proofs of [1] are now easily extended to our more general class of assertion.