Constructive Semantics

Paul C. Brown
GE Corporate Research and Development
Schenectady, New York USA

Constructive Semantics is an approach to programming language semantics that treats a program as a constructive specification for an abstract state machine. This abstract machine is composed of a set of smaller "well-behaved" machines operating concurrently. The exact combination of machines is determined by the program, with each programming language construct appearing in the program defining a portion of the composition. The programming language itself specifies a number of primitive machines that form the basic building blocks of programs. These machines represent the basic operations and data types of the language. The resulting semantics is relatively easy to understand, and its relationship to the original program is clear.

Constructive semantics treats many higher level programming language abstractions also as specifications of state machines, where these machines serve as prototypes for entire sets of machines. For example, a basic data type in a programming language is modeled as a state machine, and each variable of the type is modeled as a copy of this machine. Behavioral equivalence of machines provides a basis for modeling abstract data types, in which behaviorally equivalent machines belong to the same abstract data type. Behavioral equivalence also provides a basis for modeling type hierarchies such as those found in object-oriented languages with multiple inheritance.

The formalism underlying constructive semantics is a process algebra known as the Hybrid Calculus of Communicating Systems (HCCS), since it contains elements of both Milner's Calculus of Communicating Systems (CCS) and Hoare's Communicating Sequential Processes (CSP). Behavioral equivalence in HCCS is based upon Hoare's failures equivalence.

Constructive semantics provides straightforward semantic models for other important aspects of programming languages, including concurrency (Ada tasks), elaboration and visibility computations.

1.0 Introduction

The purpose of this paper is to show how process algebras can be used to give a complete and understandable semantics for a programming language, including such higher level language constructs as data types, program blocks, subprograms, and Ada generics. The resulting semantics is complete in the sense that *all* of the features of the programming language can be modeled in a straightforward manner[1]. The simple relationship between the programming language constructs and the semantic model elements allows an understanding of the semantics to directly aid in understanding the programming language being modeled.

We shall not attempt within the scope of this paper to give a complete model for even a small programming language. Instead, we will lay the formal groundwork for the model, and give examples of simplified semantics for several programming language

S. Purushothaman et al. (eds.), *NAPAW 92*

constructs to indicate the manner in which a complete semantics can be given. A more complete treatment of constructive semantics can be found in [5].

The following section provides an intuitive introduction to constructive semantics by giving a simple program fragment and a sketch of its corresponding semantic model. The following two sections then describe the formal underpinnings of constructive semantics, first by defining the process algebra HCCS (section 3), and then by describing the classes of machines that we will use directly in our semantics and the machine algebra that we will use for describing their composition to form larger machines (section 4). Finally, we give formal expressions for simplified semantics of some common programming language constructs.

2.0 An Overview of Constructive Semantics

The fundamental concept behind constructive semantics is that a program is simply the specification of an abstract state machine, and *all* constructs of the programming language in which the program is written relate, directly or indirectly, to the specification of this machine or its component machines. The specification for the entire program is, in turn, given as a composition of smaller state machine specifications. Each of these machines may, in itself, be a composition of still smaller machines. Since the programming language constructs themselves define the composition of machines from smaller machines, the structure of the semantic model very closely parallels the structure of the program.

Figure 1 shows some of the possible relationships between machines, using the notation of Rumbaugh et. al. [12]. Each machine is either a primitive machine or a composite machine. Composite machines are comprised of one or more machines (of either class), and, conversely, each machine is optionally a part of a composite machine. Our program itself is just a composite machine.

It is permissible to have many copies of a program executing simultaneously. Each of these copies is a state machine, separate and distinct from all of the others, yet sharing the same specification. When we ask a machine to perform an action, we must be specific as to which machine we are making the request of. To reflect this in our model, we give each action of a machine a "subscript" that uniquely identifies the machine that the action is associated with. The program is thus the specification of a *family* of *structurally isomorphic* machines (machines that are identical under a mapping replacing the action subscripts with subscripts indicating the other machine). We will call a family of structurally isomorphic machines a ***state type*** (for reasons that will become apparent later).

1. The only exception that we are aware of is that the concept of a time *interval*, such as the interval implicit in the Ada `delay` statement, is not directly definable, but must be defined with reference to some abstract clock, each of whose "ticks" is implies the passage of a certain time interval. This is a consequence of the underlying process algebra (as are all process algebras that we are aware of) being based upon a *point* ontology of time rather than an *interval* ontology. [13] contains an extensive discussion and comparison of these ontological structures.

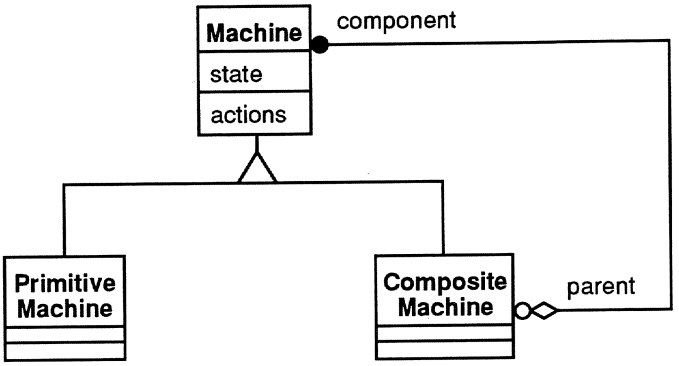

Figure 1 Composition of Machines

This concept of families of machines characterizes other programming language constructs as well. Subprograms and operators can be interpreted as families of state machines: each call of the subprogram or operation indicates an interaction with a different machine from the family. Data types are also families of machines, each of which duplicates the data storage and behavior associated with the type. Each variable belonging to a data type is associated with one member of the family of machines defined by the data type.

2.1 Behavioral Equivalence

As might be surmised, for any given behavior there are many models (machines or combinations of machines) that will exhibit that behavior. In constructive semantics, we use Hoare's failures equivalence [8][2] to determine when two machines are equivalent. Under failures equivalence, machines are considered equivalent if the sequences of actions that they will perform are the same and the actions that they can possibly refuse after a sequence of actions are also the same.

Since *implementations* of programs (as opposed to their abstract semantics) can also be modeled with process algebras, behavioral equivalence provides us with a formal means of comparing actual implementations of programs with the abstract semantics. Furthermore, since this behavioral equivalence is a congruence relation in the model, equivalence of two machines can be established by establishing that the component pieces of the two machines are pair-wise equivalent.

2.2 Types

There are at least three concepts frequently associated with the term type in programming languages: a set of values[2], a behavioral (interface) specification, or an implementation specification. In constructive semantics, these three concepts are not inconsistent, and are in fact closely related to one another. To see this, we must first realize that in a state machine model, there is no way to store a value directly. Instead, the value must be encoded as the state of some state machine, with each state of this

2. Or a representative of the set, as is the case in domain semantics

machine representing a different value. Reading and writing values then correspond to actions of the state machine in which the state of the machine is altered (writing) or its present state is revealed without altering the state (reading). Thus we see that values, per se, do not exist in our model. We must instead talk in terms of a machine capable of encoding these values. In particular, in order to model the storage or passing of values, we must explicitly model the machine that is used to hold the values. This explicit handling of value storage in the semantics makes clear the distinctions between the different parameter passing semantics used in subprograms, and provides a basis upon which to decide whether they are equivalent for a particular subprogram.

When a machine is interacting with a machine encoding values (we will call these machines ***value machines***), we are not *directly* observing the state of the machine (i.e. the encoding of the value being represented): we are only able to *infer* the intended value from the actions to which the value machine will respond. We conclude that the only characteristic we *can* observe about a machine is its *behavior*: its interactions with other machines. Returning to our discussion concerning a type as a set of values, it is the behavior of the machine that we use to store that values rather than the values themselves that we find in constructive semantics: values have thus been reduced to a special case of a type as a behavioral specification. We define an ***abstract type*** to be a set of actions and a family of machines that all exhibit the same behavior with respect to the actions belonging to the abstract type. Clearly more than one state type may be included in a given abstract type, and is relatively straightforward to show that every state type also defines an abstract type. Similarly, we define a ***state type*** to be a set of machines and an isomorphism between the actions of the machines. Thus all of the machines in a state type are identical up to the relabeling operation.

In constructive semantics, a data type declaration in the language is usually taken to be both the definition of an abstract type (a behavioral specification) and a state type belonging to that abstract type (an implementation) capable of encoding the values of the type. For built-in data types in the language, the state type and abstract type are taken to be part of the language specification itself. For user-defined types, the formal semantics of the language defines how these types are defined in terms of previously defined abstract and state types.

Abstract types provide a formal basis for defining type hierarchies based upon behavioral equivalence, in which the descendent types are required to exhibit the full abstract behavior of the parent[3]. This allows the implementation of the individual types to be different, and allows their functionality to be extended (new actions added) beyond that of the parent as long as the behavior with respect to the parent's actions is preserved. Abstract types even provide a model for multiple inheritance. In Figure 2 child type **C** performs all "a" actions and "b" actions as well as its own "c" actions. If the "a" actions and "b" actions are disjoint (a condition that we require of all unrelated types for both technical and philosophical reasons), the child type **C** will be of abstract type **A** (behaviorally equivalent to **A** when interaction is restricted to

3. These are frequently referred to as "is-a" type hierarchies in object oriented languages.

"a" actions) and will also be of abstract type ***B*** (behaviorally equivalent to ***B*** when interaction is restricted to "b" actions).

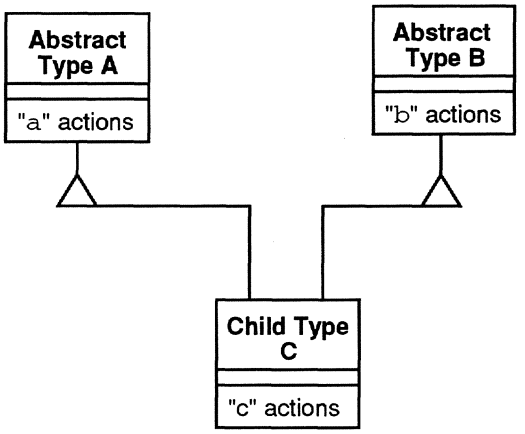

Figure 2 Multiple Inheritance

In constructive semantics, we will define families of machines by specifying a prototype machine for the family. The other machines in the family can then be generated by changing the subscripts of the actions as appropriate. This holds for abstract types as well as state types.

2.3 Classes of Machines

There are three classes of primitive machines that occur in constructive semantics. The first of these is the class of ***value machines***, which we have already encountered. Value machines are passive (by construction): they never ask other machines to perform actions. Thus value machines cannot interact.

Since value machines never request actions of other machines, and therefore cannot interact, we must have at least one other class of machines that can request actions of other machines. In fact, we have two classes of these machines. The first we call ***interaction machines***. Generally speaking, interaction machines do not retain encoded values in their internal state: instead, they act as intermediaries between other machines that can encode values. These are the machines that define the basic operations and relations: assignment, equality, arithmetic operations, etc.

The third class of machine correlates the activation and idling (starting and stopping) of other machines (we require that each machine has a unique activate and idle action). This class of machines we will call ***activation machines***.

2.4 An Example

Let us consider the following fragment as if it were a complete program, and examine its possible composition from smaller machines. The data type `integer` corresponds to a state type of value machines, and the machines representing the variables `a`, `b` and `c` are members of this state type. We will use $\mathbf{V}_a$, $\mathbf{V}_b$, and $\mathbf{V}_c$ to represent these

three machines. In addition, we will use a fourth temporary variable V_t to hold the result of the addition prior to its assignment to the variable c.

```
declare
    a : integer := 2;
    b : integer := 3;
    c : integer;
begin
    c := a + b;
end;
```

Example 1 Simple Program Fragment

Constants are also represented by value machines (variants of the integer value machine), and we will use C_2 and C_3 to represent the machines corresponding to the integer constants 2 and 3 respectively. The operations of integer assignment and addition correspond to families of interaction machines. We will designate individual interaction machines with a subscript, and use a superscript notation to informally indicate the role that the machine plays in the program. For example, the first assignment machine (the one that assigns the constant 2 to a) would be designated by $=_1^{2\rightarrow a}$, and similarly the other assignment machines are designated by $=_2^{3\rightarrow b}$ and $=_3^{t\rightarrow c}$. The addition machine is designated by $+^{a,b\rightarrow t}$ (we omit the subscript here since there is only one addition machine involved).

Figure 3 shows the relationship between these machines using a reduced Petri net notation. The tokens in this Petri net are the individual machines that we have been discussing. Arcs originating at the heavy black bars indicate the activation of machines (the introduction of that machine or token into the network), and the labels indicate which machine is being activated. Arcs terminating at the heavy black bars indicate the idling of machines (the removal of machines or tokens from the network). The heavy black bars are, themselves, graphic representations of activation machines. In all of the other transitions, the tokens retain their individual identities as they pass through the transition.[4]

In this diagram, we see that the three value machines V_a, V_b, and V_c, the two constant machines C_2 and C_3, and the first assignment machine $=_1^{2\rightarrow a}$ are all activated in parallel. When the first assignment machine is idled, the second assignment machine $=_2^{3\rightarrow b}$ is activated. When the second assignment machine is idled, both the temporary value machine V_t and the addition machine $+^{a,b\rightarrow t}$ are activated. The idling of the addition machine activates the final assignment machine $=_3^{t\rightarrow c}$, and finally the four value machines, two constant machines and the final assignment machine become idle simultaneously.

While this graphical notation is quite descriptive of the relationships between the machines, it is somewhat cumbersome. Consequently, we will use an algebraic notation that also describes the machine relationships. The expression for the above example is:

4. This extension to Petri nets in which token identity is preserved through the transition is due to Eaker [6][7]

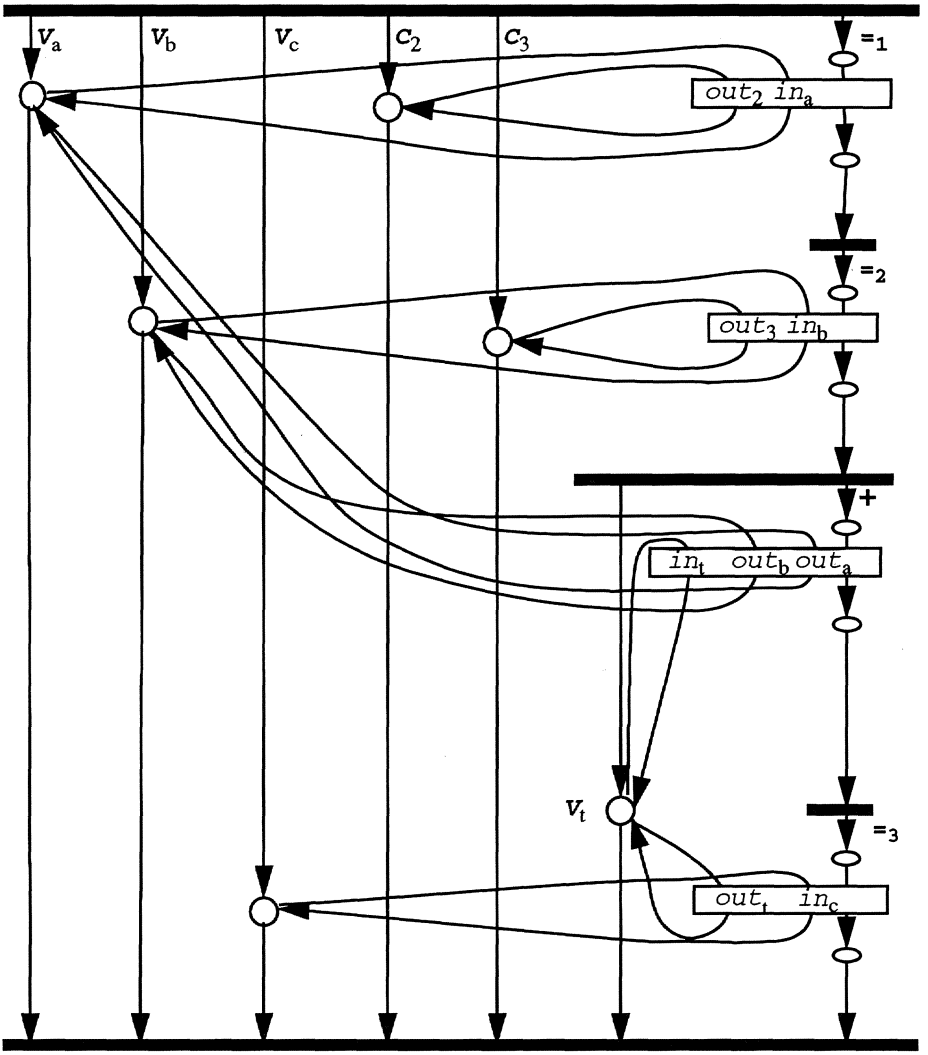

Figure 3 Reduced Petri Net of Example 1

$$\langle V_a \mid V_b \mid V_c \mid C_2 \mid C_3 \mid =_1^{2\to a} ;=_2^{3\to b} ;\langle V_t \mid +^{a,b\to t};=_3^{t\to c} \rangle\rangle \qquad (1)$$

Here the vertical bars, angle brackets and semicolons represent specific kinds of activation machines. The vertical bars | indicates parallel composition, with the angle brackets ⟨•••⟩ indicate that the parallel machines within the brackets are both activated and idled simultaneously. The semicolon indicates that the idling of the machine before the semicolon is associated with the activation of the machine after the semicolon (we adopt the convention that the semicolon binds more tightly than the parallel composition bar, and thus avoid the need for parentheses to explicitly group terms). Note, however, that in this notation there is no explicit indication of

which machines interact with which other machines. This information is contained in the definitions of the machines themselves, as we shall see in later sections.

Ada generics and C++ templates are modeled as partial specifications of state machines. These partial specifications contain variables corresponding to the formal parameters of the generic or template. The expected values for these variables are state machines. An instantiation of the generic or template is modeled as the state machine defined by replacing each variable with the state machine corresponding to the actual parameter (usually the prototype machine associated with a type or subprogram).

3.0 HCCS

We now proceed to give the formal underpinnings to constructive semantics. HCCS is the process algebra upon which constructive semantics is based, and is a hybrid of work done by Milner, Hoare and Brookes. The formal basis for this model is the synchronization tree semantics of Brookes [2].

The starting point for the semantic model is Milner's CCS [9][10], with the following modifications:

1 Instead of using CCS's flat set of actions, we use the abelian group of actions as used in Milner's SCCS and ASCCS.

2 Milner's + operator is replaced by the similar $\Box$ operation used by Hoare and defined in terms of Milner's synchronization trees by Brookes [2]. We will use $\oplus$ to represent this operation. The use of the Hoare operator makes failures equivalence a congruence relation.

3 We alter the definition of Milner's | operation to allow n-way synchronization between agents, where CCS only allows binary synchronization. We provide a formal semantics for this new operation using Brookes' synchronization tree semantics.

4 We use Brookes' failures equivalence rather than Milner's observational equivalence. Milner's observational equivalence makes non-observable distinctions between machines, while Brookes' failures equivalence only distinguishes between machines if the machines differ in observable behavior. The use of failures equivalence allows the proof of our parallelization theorem, which is not true using observational equivalence.

3.1 Actions

We first define `Act` the set of ***actions***. Actions can be thought of as the labels on the arcs of state machines. Actions are part of an abelian group $(\mathtt{Act}, \times, 1, \overline{\ })$ freely generated by a set of *positive primitive actions* = {a,b,c,...}, a unique identity action 1, a binary operator $\times$ and a unary inverse operator $\overline{\ }$. The inverse of a positive primitive action a is the *negative primitive action* $\overline{a}$ (and vice-versa). We shall refer to the union of the positive and negative primitive actions and the identity action as the set of *primitives* = $\{...,\overline{c},\overline{b},\overline{a},1,a,b,c,...\}$.

In our subsequent semantics, we shall see that the operator × is used as a parallel composition operator: when we write a×b, we mean that the actions a and b occur simultaneously. In writing actions, we shall frequently omit the parallel composition operator ×, writing st to represent s × t.

To facilitate the comparison of actions, we define *Primitives*(a) to be the set of primitive actions that an action a is comprised of. For example, if a and b are primitive actions, then *Primitives*(ab) = {a, b}. We note that this is a definition based on the *syntactic structure* of the action as a minimum length string of primitive actions. In particular, we do *not* want to infer that $\{a, \bar{a}\} \subseteq$ *Primitives*(x) just because $a\bar{a}=1$ and 1x = x for all actions $x \in Act$. However, we will always include the element 1 in the set of actions generated by *Primitives*(a). By extension, if A is a *set* of actions, we define *Primitives*(A) $= \cup_{a \in A}$ *Primitives*(a).

From time to time we will wish to define the set of actions a^+ that can be constructed from the primitive actions *Primitives*(a) of an action a. Formally, we define a^+ to be the submonoid of Act freely generated by *Primitives*(a) for any $a \in Act$.[5] As before, we extend this to sets of actions, defining A^+ to be the submonoid of Act freely generated by *Primitives*(A) for any $A \subseteq Act$. In a similar manner, we define the subgroup of actions a* that can be constructed from the primitive actions *Primitives*(a) of an action a. Formally, we define a* to be the subgroup of Act freely generated by *Primitives*(a) for any $a \in Act$.We extend this to sets of actions, defining A^* to be the subgroup of Act freely generated by *Primitives*(A) for any $A \subseteq Act$.

3.2 Agents and Agent Expressions: Syntactic State Machine Specifications

We now proceed to give a syntax for the specification of a state machine. We begin by defining $\boldsymbol{E}$, the set of ***agent expressions***. Agent expressions are syntactic representations of the states of state machines. Agent expressions may contain variables (whose values will be other agent expressions), in which case the agent expression is a *template* for a state that will be fully specified when the values of the variables appearing in the agent expression are given. An ***agent*** is an agent expression that contains no variables. An agent can be viewed as a fully specified state of a state machine. As we shall see, an agent, in conjunction with a collection of constant definitions and the HCCS inference rules relating one agent expression (state) to another, provides a complete specification of a state machine.

Returning to the syntactic specification of agent expressions, let $\boldsymbol{K} = \{0,1,A,B,C,...\}$ be a set of ***agent constants***, and $\boldsymbol{X} = \{X,Y,Z,...\}$ be a set of ***agent variables***. Then we define $\boldsymbol{E}$, the set of ***agent expressions***, to be the set generated by the following rules:

- $\boldsymbol{K} \cup \boldsymbol{X} \subset \boldsymbol{E}$

$\forall E,F \in \boldsymbol{E}, a \in Act, A \subseteq Act$:

Action:

5. Since a monoid does not include the inverse operator, the only primitives in A^+ are those that explicitly appear in *Primitives*(A).

$a.E \in \boldsymbol{E}$

where "." is a binary operator of type $Act \times \boldsymbol{E} \rightarrow \boldsymbol{E}$. Informally, this operator is used to provide the semantics of sequence: the agent a.E (under appropriate circumstances) performs the action a and then behaves like E.

Product:

$E \mid F \in \boldsymbol{E}$

where "|" is a binary operator of type $\boldsymbol{E} \times \boldsymbol{E} \rightarrow \boldsymbol{E}$. Informally, this operator is used to provide the semantics of parallel composition: E | F means that both E and F are operating in parallel.

Summation:

$$\sum_{i \in I} E_i \in \boldsymbol{E}$$

where "$\sum_{i \in I}$" is an n-ary operator of type $\boldsymbol{E} \times \boldsymbol{E} \times \ldots \times \boldsymbol{E} \rightarrow \boldsymbol{E}$. Informally, this operator is used to provide the semantics of deterministic choice between the behaviors of the various E_i in the term. In the special case of a summation involving two machines, we shall write the summation as $E \oplus F$.

It is important to note that the operation being defined here is *not* Milner's summation operator, but a generalization to an arbitrary number of terms of Hoare's [] operator. The behavior of the Milner and Hoare operators is the same for observable actions, but different for the non-observable action 1.

Restriction:

$E \upharpoonright A \in \boldsymbol{E}$

where "$\upharpoonright$" is a binary operator of type $\boldsymbol{E} \times A \rightarrow \boldsymbol{E}$, where A is a subset of Act. Informally, this is a restriction operator, with the semantics that $E \upharpoonright A$ restricts the visible (available) actions of E to those present in A^* (i.e. hides all actions of E that are not in A^*).

From this operator we define a derived binary operator $\backslash : \boldsymbol{E} \times A \rightarrow \boldsymbol{E}$ as follows:

$$E \backslash A \equiv E \upharpoonright \{a \in Act \mid \mathit{Primitive}(a) \cap \mathit{Primitive}(A^*) = 1\}$$

This operator hides any actions any of whose primitive actions are also primitive actions of A^*.

Morphism:

$E[\phi] \in \boldsymbol{E}$

where [] is a binary operator of type $\boldsymbol{E} \times (Act \rightarrow Act) \rightarrow \boldsymbol{E}$, and $\phi : Act \rightarrow Act$ is any mapping from Act to Act such that $\phi(\overline{a}) = \overline{\phi(a)}$ and $\phi(1) = 1$. Informally, [] is a relabeling operation such that $E[\phi]$ is the agent expression that results from replacing each action of E with the result of applying the mapping ϕ to that action.

Constant Definitions:

To allow recursive definitions, we use systems of equations involving constants, where a constant is simply a variable whose value has been fixed to be a particular agent ex-

pression. Recursive definitions can then be achieved by the appropriate use of constants on the right hand side. Each constant is defined to be an agent expression E:

$$A \equiv E$$

3.3 Semantics: Labeled Transition Systems

We now proceed to give semantics to agent expressions by defining *labeled transition systems*. Informally, a labeled transition system is a state machine in which the agent expressions are the "states" of the machines, and the transitions between agent expressions are labeled with actions, hence the term labeled transition system. Formally, a ***labeled transition system*** is a triple:

$$(\boldsymbol{E},\ Act,\ \{\xrightarrow{a} : a \in Act\})$$

where $\boldsymbol{E}$ is a set of agent expressions, Act is a set of actions, and each $\xrightarrow{a}$ is a relation between agent expressions.

In this system, we have the following inference rules[6]

Action:

$$\frac{}{a.E \xrightarrow{a} E} \tag{2}$$

This rule states that the agent expression a.E becomes the agent E after it has performed the action a.

Summation:

$$\frac{E_j \xrightarrow{a} E'_j}{\sum_{i \in I} E_i \xrightarrow{a} E'_j} \ (j \in I, a \neq 1) \tag{3}$$

This rule states that if any element E_j of the summation can make a transition to E'_j by performing the observable action a, then the entire summation can also perform the a action and then behave like E'_j. In other words, by performing this observable action, all of the other alternative choices have been discarded.

$$\frac{E_j \xrightarrow{1} E'_j}{\sum_{i \in I} E_i \xrightarrow{1} \sum_{i \in I} E_i\,[E'_j/E_j]} \ (j \in I) \tag{4}$$

This second rule deals with the case in which the action is the non-observable action 1. In this case, the individual component of the summation may make this non-observable transition, *but the other choices in the summation are not discarded.* The notation $[E'_i/E_i]$ simply indicates that E_i has been replaced by E'_i.

6. While they are given here as axioms, these axioms are in fact derivable from synchronization tree semantics [5]

This pair of rules has the effect of preventing a non-observable transition of one element of a summation from eliminating the other possibilities in the summation. This is distinctly different from Milner's "+" semantics[7], which uses the first rule only and places no restriction on the observability of the action. Using Milner's semantics would prevent failures equivalence from being a congruence relation, which is a requirement of our model[8].

Product:

$$\frac{E \xrightarrow{a} E'}{E \mid F \xrightarrow{a} E' \mid F} \tag{5}$$

$$\frac{F \xrightarrow{b} F'}{E \mid F \xrightarrow{b} E \mid F'} \tag{6}$$

These two rules state that if one member of a parallel composition can make a transition on the action a, then that member can also make the transition as part of a parallel composition if the parallel composition performs the action a.

$$\frac{E \xrightarrow{a} E' \qquad F \xrightarrow{b} F'}{E \mid F \xrightarrow{ab} E' \mid F'} \tag{7}$$

This rule says that both members of a parallel composition may make transitions simultaneously, provided that the appropriate actions occur simultaneously.

Restriction:

$$\frac{E \xrightarrow{a} E'}{E \restriction S \xrightarrow{a} E' \restriction S} \ (a \in S) \tag{8}$$

Here we have the semantics of restricting the actions that a machine may perform to those present in a particular set S. If E makes a transition to E′ with the action a, and a is a member of S, then $E \restriction S$ may also make a transition to $E' \restriction S$ with the action a. Note that the absence of any other rule regarding restriction means that if a is not in S, no transition is possible. We also note that the non-observable action 1 is always a member of S.

Morphism:

7. [9] p. 271
8. Milner's observational equivalence is also not a congruence relation in CCS. It is conjectured that the use of Hoare's □ operator (which we are using here as our + operator) in lieu of Milner's + would make observational equivalence a congruence relation in the modified CCS system.

$$\frac{E \xrightarrow{a} E'}{E[\phi] \xrightarrow{f(a)} E'[\phi]} \quad (9)$$

This rule gives us the semantics of relabeling machines. Recall that ϕ is a homomorphism from actions to actions. The semantics are that if E can make a transition to E′ with the action a, then the relabeled version of E, E[ϕ], can make a transition to the relabeled version of E′, E[ϕ]′, with the mapped action ϕ(a).

Constants:

$$\frac{E \xrightarrow{a} E'}{C \xrightarrow{a} E'} \ (C \equiv E) \quad (10)$$

The semantics of constant declarations is that if the constant C is defined to be the expression E, then any transition that E can make is also a transition of C.

4.0 Machines

In constructive semantics, a machine is a labeled transition system and an initial state of that system. The well-behaved machines used in constructive semantics are a restricted subset of the machines definable in HCCS. We now proceed to define the restrictions that we will place upon the machines, and then define the machine algebra in which machines may be composed in various serial/parallel combinations while preserving these well-behaved properties. Several theorems in machine algebra describe orthogonality (independence) conditions under which the serial/parallel relationships between machines in a composition may be altered, yielding behaviorally equivalent compositions.

4.1 Positive Primitive Actions Appear on Exactly One Machine

We now proceed to give an interpretation of actions. Positive actions represent actions that a machine can take. Since we wish such actions to be unique to a machine, we shall require that ***a positive primitive action may appear as a transition label on exactly one machine*** (although it may label any number of arcs on that machine). In contrast, the corresponding negative primitive action (a request made by another machine for this action) may appear on any number of machines. We will refer to the one machine on which a positive action appears as the ***defining machine*** of that action.

4.2 Well-Behaved Machines

We define a ***well-behaved machine*** to be a machine that has exactly one observable activate action and whose initial action is always that unique activate action, and has exactly one observable idle action and if this idle action occurs, the only observable action that may follow is the activate action[9].

9. Note that this does not imply that the machine must respond to the idle action in every state. This is similar to Milner's well-terminating property [10] p. 173

We note that ";" and "⟨⟩" both preserve the well-behaved property:

Lemma 1

Let $\mathcal{M}$ be the set of well-behaved machines. Then:

$$\mathbf{M}_1, \mathbf{M}_2 \in \mathcal{M} \Rightarrow \mathbf{M}_1;\mathbf{M}_2 \in \mathcal{M}$$

$$\mathbf{M}_1, \mathbf{M}_2, \ldots, \mathbf{M}_n \in \mathcal{M} \Rightarrow \langle \mathbf{M}_1 | \mathbf{M}_2 | \ldots | \mathbf{M}_n \rangle \in \mathcal{M}$$

We adopt the convention that the ";" operator binds more tightly than (takes precedence over) the | operator, and that ⟨⟩ binds more tightly than either.

4.3 Machine Algebra

Our ***machine algebra*** is then the system $(\mathcal{M}, =, ;, |, \langle\rangle)$, where $\mathcal{M}$ is the set of well-behaved machines, = is behavioral equivalence, and ;, |, and ⟨⟩ are as defined above.

4.3.1 Parallelization Theorem

We wish to consider the circumstances under which the sequencing of machines in a composition may be altered. Consider the following abstracted model of a program or subprogram:

$$\langle \mathbf{V}_a \mid \mathbf{V}_b \mid \mathbf{M}_p; \mathbf{M}_q ; \mathbf{M}_r ; \mathbf{M}_s \rangle \backslash \mathcal{S}$$

where $\mathcal{S} = Sort^{10}(\mathbf{M}_a) \cup \overline{Sort}(\mathbf{M}_a) \cup Sort(\mathbf{M}_b) \cup \overline{Sort}(\mathbf{M}_b) \cup Sort(\mathbf{M}_q) \cup \overline{Sort}(\mathbf{M}_q) \cup Sort(\mathbf{M}_r) \cup \overline{Sort}(\mathbf{M}_r)$, $\mathbf{V}_a$ and $\mathbf{V}_b$ are local variables of the program, $\mathbf{M}_p$ is a machine that copies actual parameter values into the local variables, $\mathbf{M}_q$ and $\mathbf{M}_r$ are machines that do the actual work of the subprogram by modifying the local variables, and $\mathbf{M}_s$ is a machine that copies the final values out into their target destinations. The restriction $\backslash\mathcal{S}$ simply says that the internal workings of the program are not visible from outside the program (the variables are hidden and the machines that do the work cannot communicate with any machines outside of the program). The sequencing $\mathbf{M}_p; \mathbf{M}_q ; \mathbf{M}_r ; \mathbf{M}_s$ says that the actual parameter values are copied in, then $\mathbf{M}_q$ does its work, then $\mathbf{M}_r$ does its work, and finally $\mathbf{M}_s$ copies the results back out of the program.

Now it seems intuitive that if 1) the variables cannot interact with each other, and 2) the machines that do the work ($\mathbf{M}_q$ and $\mathbf{M}_r$) cannot interact with each other, and 3) each working machine can only interact with *one* of the variables, then it should not matter which order $\mathbf{M}_q$ and $\mathbf{M}_r$ do their work. In fact, they could even operate in parallel! This is exactly what the parallelization theorem establishes:

Theorem 2 Parallelization Theorem

$$\mathbf{M}_a \perp \mathbf{M}_b, \mathbf{M}_a \perp \mathbf{M}_r, \mathbf{M}_b \perp \mathbf{M}_q, \mathbf{M}_q \perp \mathbf{M}_r \Rightarrow$$
$$\langle \mathbf{M}_a \mid \mathbf{M}_b \mid \mathbf{M}_p ; \mathbf{M}_q ; \mathbf{M}_r ; \mathbf{M}_s \rangle \backslash \mathcal{S} =$$
$$\langle \mathbf{M}_a \mid \mathbf{M}_b \mid \mathbf{M}_p ; \langle \mathbf{M}_q \mid \mathbf{M}_r \rangle ; \mathbf{M}_s \rangle \backslash \mathcal{S} =$$
$$\langle \mathbf{M}_a \mid \mathbf{M}_b \mid \mathbf{M}_p ; \mathbf{M}_r ; \mathbf{M}_q ; \mathbf{M}_s \rangle \backslash \mathcal{S}$$

10. The sort of a machine is the set of primitive actions that appear on the machine. The inverse of the sort is the set of inverses of the actions appearing in the sort.

This is a valuable result, since it shows how to take a serial program and convert it into an equivalent parallel program with no analysis beyond simply determining the orthogonality (independence) of the component parts of the program. The proof of this theorem is in [5].

4.3.2 Change of Scope Theorem

Somewhat similar to the parallelization problem is the change of scope theorem. Consider the following configuration of machines:

$$\langle \mathbf{M}_a ; \langle \mathbf{M}_b \mid \mathbf{V}_x \rangle ; \mathbf{M}_c \rangle \backslash S$$

where $S = Sort(\mathbf{V}_x)$. The situation we are modeling here is one in which $\mathbf{V}_x$ is a local variable used by $\mathbf{M}_b$ only. If $\mathbf{M}_a$ and $\mathbf{M}_c$ cannot interact with $\mathbf{V}_x$ and $\mathbf{V}_x$ is hidden from the outside, then it seems reasonable that the lifetime of $\mathbf{V}_x$ could be extended, yielding a behaviorally equivalent configuration:

$$\langle \mathbf{V}_x \mid \mathbf{M}_a ; \mathbf{M}_b ; \mathbf{M}_c \rangle \backslash S$$

This leads to the following theorem:

Theorem 3 Change of Scope Theorem

$$\mathbf{M}_a \perp \mathbf{M}_x, \mathbf{M}_c \perp \mathbf{M}_x \Rightarrow$$
$$\langle \mathbf{M}_a ; \langle \mathbf{M}_b \mid \mathbf{M}_x \rangle ; \mathbf{M}_c \rangle \backslash S = \langle \mathbf{M}_x \mid \mathbf{M}_a ; \mathbf{M}_b ; \mathbf{M}_c \rangle \backslash S$$

where $S = Sort(\mathbf{M}_X)$. A proof of this theorem can be foundin [5].

5.0 Semantics

The intent of constructive semantics is to provide an interpretation of a program as the specification for an abstract state machine. We are now in a position to show how the constructive semantics of a programming language can be given using the results of the earlier sections. The style of this semantics will be denotational, showing how each construct in the language can be interpreted as a machine that is defined by the composition of the denotations of its component parts.

A programming language defines a number of basic data types and operations as given elements of the language. We assume that the machines denoted by these data types and operations are given as part of the formal semantics of the language.

We will take some example from the Ada programming language as the basis for showing how a constructive semantics for a language can be given. This subset includes many of the language features of Ada, including declarative blocks, composite types and exception handling. Packages and tasks have been omitted because they differ little from declarative blocks in their semantics except for their extremely complex visibility computation rules, which are analyzed and modeled in [4].

In giving the semantics for each construct, we will not provide complete Ada semantics, but rather simplified semantics that highlight the strategy used in modeling each particular language construct. This is done because modeling the complex visibility rules of Ada would tend to divert attention away from the basic approach.

5.1 Declarations and References

In the previous sections we have laid the groundwork for modeling the semantics of a programming language in terms of machines. Each identifier in a program corresponds to a machine. We define an ***environment*** $\mathcal{E}: Ide \times \mathcal{M}$ to be a relation between identifiers and machines. We note that this environment is suitable for mapping identifiers into both machine instances and types if we use the "prototype machine" approach to defining machine types.

A ***declaration*** in a program usually results in both the definition of a machine and its entry in an environment relation in association with an identifier[11]. A ***reference*** is an occurrence of an identifier that must be associated with a machine through an environment relation[12]. *Each occurrence of an identifier in a program is either part of a declaration or it is part of a reference.* Consider the program in Example 2. In this example, we observe a single explicit declaration[13] of a variable named `a`, and a number of references: one to `integer` in the declaration itself; another to `a` in the assignment statement; and a third to the literal `1` in the assignment statement. The symbol `:=` is a reference to an assignment operator.

```
X: declare
       a : integer; -- a declaration of "a"
   begin
       a := 1; -- a reference to the "a"
   end X;
```

Example 2 Declarations and References

Taking this perspective of a program, one interesting problem is the determination of which machine a particular reference actually refers to. This problem can, in itself, be divided into two sub-problems: one is the determination of which machines are *visible* (available) at a given point in the program (we call this set of visible machines and their associated identifiers the ***direct environment***); the other is the determination of which of these is the one that is actually being referred to. In [4] we have investigated the computation of visibility, and at present have left the formalization of how a reference is selected for future work.

5.2 Elaboration: From Programs to Machines

If a program is the specification of a state machine, somewhere along the line the program must be converted into the machine itself. While one might be initially tempted to say that this is the role of the compiler, a compiler in simply generates the *initial state* of a machine, namely the computer in which the program will be executed. In constructive semantics, we wish to generate the specification of an abstract machine whose behavior is the "meaning" of the program. Of course, to be correct, the behavior

11. Note that this approach allows more than one identifier to be associated with the same machine. Declarations of aliases do not define new machines, but simply associate an existing machine with a new identifier.
12. There may be more than one entry in the relation with the same identifier
13. There is also an implicit declaration of the block `X`.

of the computer with the compiler generated initial state must be equivalent to the behavior of this abstract machine.

Converting a program into a state machine is not necessarily a one step process. The obvious counter-example is the interpreter, which does the conversion piecemeal. But even a compiled program may not be convertible into a complete state machine at load time. For example, some Ada data type declarations (definitions of machine types) are allowed to depend upon values computed previously in the program. Thus the machines defined by these type declarations are not even fully defined until part of the program has been executed.

In what follows, we shall have occasion to refer to the *process* of converting a program into a machine. For this purpose, we borrow a term from Ada and call this process ***elaboration***. Formally, elaboration is a mapping from a syntactic language term, a local declaration set, a direct environment and a type structure to a machine:

$$\mathbf{E}: \mathcal{L} \times \mathcal{E} \times \mathcal{E} \times \mathcal{T} \rightarrow \mathcal{M}$$

The first term is the language expression whose meaning is being determined. The second term is an environment that is to contain local declarations (typically, this environment is modified in the course of elaboration). The third term is an environment that contains machines that have been defined elsewhere that may be used in the course of performing the elaboration.The fourth term is a type structure, which may contain some type information initially and may also be added to during the course of elaboration.

5.3 Variables of a Simple Type

A variable belonging to a simple data type is modeled as a value machine belonging to the state type whose prototype machine is associated with the data type. The result of elaboration is a new machine "cloned" from the prototype machine of the data type.

$$\mathbf{E}([\![\ \texttt{variablename: typename;}\ ]\!], \mathbf{LD}, \mathbf{DE}, \mathcal{T})$$
$$= \mathbf{M} \equiv \mathbf{New}(\mathbf{M}_T)$$

where

$$\mathbf{M}_T = \mathbf{Ref}(\texttt{ide}, \mathbf{DE})$$

Here **New** is an operation that takes a prototype machine and gets an unused member of the state type, and **Ref** is an operation that locates a machine in an environment by name.

The elaboration has side effects adding the new declaration to the set of local declarations and recording the type relationship in the *StateType* relation:

$$\mathbf{LD} = \mathbf{LD} \cup \{(\texttt{variablename}, \mathbf{M})\}$$

$$\mathit{StateType} = \mathit{StateType} \cup (\mathbf{M}, \mathbf{M}_T)$$

where *StateType* is a relation that records which state type a machine belongs to.

5.4 Subprogram Calls and Expression Evaluation

The semantics that we give for subprogram calls encompasses the invocation for both operators and subprograms. This treatment of operators as pre-defined subprograms allows the modeling of languages in which user-defined versions of these operators may be declared in a program. The newly defined operator affects the visibility of the original operator according to the visibility rules of the language.

We give several alternative representative formulations for calling two argument subprograms, each of which can be readily generalized to an arbitrary number of arguments. In defining these expressions, we will frequently need to know the syntactic name of the operation at the root of an expression. We define the syntactic function $\mathcal{R}oot$(expression) that returns this identifier.

The simplest form of subprogram call assumes that the arguments are simply references to existing machines (thus requiring no elaboration of the arguments) and further assumes that the arguments are of the correct type. We arrive at a relatively simple semantic for a subprogram call in which the prototype machine defining the subprogram is located and a new member of its associated state type is returned:

$$\mathbf{E}([\![\ \texttt{subprogramName (<argument}_1\texttt{>, <argument}_2\texttt{>)}\ ;\ ]\!], \mathbf{LD}, \mathbf{DE}, \mathcal{T})$$
$$= \mathbf{New}(\mathbf{Ref}(\texttt{subprogramName}, \mathbf{DE}))[\mathtt{M}_1/\mathtt{P}_1, \mathtt{M}_2/\mathtt{P}_2]\ \rangle$$

where $\mathtt{M}_1 = \mathbf{Ref}(\mathcal{R}oot(\texttt{<argument}_1\texttt{>}))$, $\mathtt{M}_2 = \mathbf{Ref}(\mathcal{R}oot(\texttt{<argument}_2\texttt{>}))$, and $[\mathtt{M}_1/\mathtt{P}_1, \mathtt{M}_2/\mathtt{P}_2]$ is a minor abuse of our morphism notation indicating that the actions of the formal parameter $\mathtt{P}_1$ are mapped to the actions of actual parameter $\mathtt{M}_1$, and similarly for $\mathtt{P}_2$ and $\mathtt{M}_2$. This assumes, of course, that $\mathtt{P}_1$ and $\mathtt{M}_1$ are of the same type, and similarly $\mathtt{P}_2$ and $\mathtt{M}_2$ are of the same type.

We now extend the semantics to include a modest amount of type checking. In this scheme, type information propagates only in one direction: upwards from actual arguments to functions.

$$\mathbf{E}([\![\ \texttt{subprogramName (<argument}_1\texttt{>, <argument}_2\texttt{>)}\ ;\ ]\!], \mathbf{LD}, \mathbf{DE}, \mathcal{T})$$
$$= \mathbf{New}(\mathbf{TypRef}(\texttt{subprogramName}, \mathbf{DE},$$
$$\{\mathbf{RetSig}(\mathtt{M}_1) \times \mathbf{RetSig}(\mathtt{M}_2)\}))[\mathtt{M}_1/\mathtt{P}_1, \mathtt{M}_2/\mathtt{P}_2]$$

Here **RetSig** uses the type structure $\mathcal{T}$ to locate the return type signature of the indicated machine, which is, itself, a prototype machine representing some data type, and **TypRef** uses the same type structure to type qualify candidate machines found in the environment.

The semantics given for the previous two examples will work correctly if the argument is a variable, but if the argument is itself a subprogram reference (a function call), then the reference to the root of the argument will return the *prototype* machine of the subprogram, and we must then get a new member of its state type. We do this by elaborating the argument itself as part of the elaboration of the subprogram call. We thus get (ignoring type checking again):

$$\begin{aligned}
&\mathbf{E}(\llbracket \texttt{subprogramName (<argument}_1\texttt{>, <argument}_2\texttt{>);} \rrbracket, \mathbf{LD}, \mathbf{DE}, \mathcal{D}) \\
&= \langle\, \mathbf{New}(\mathbf{Ref}(\texttt{subprogramName}, \mathbf{DE}))[\mathbf{M}_1/\mathbf{P}_1, \mathbf{M}_2/\mathbf{P}_2] \mid \\
&\quad \text{if } \mathbf{Type}(\mathbf{M}_1) = \mathbf{M}_1 \text{ then } \mathbf{E}(\llbracket \texttt{<argument}_1\texttt{>} \rrbracket, \mathbf{LD}, \mathbf{DE}, \mathcal{D}) \text{ else } \mathbf{0} \mid \\
&\quad \text{if } \mathbf{Type}(\mathbf{M}_2) = \mathbf{M}_2 \text{ then } \mathbf{E}(\llbracket \texttt{<argument}_2\texttt{>} \rrbracket, \mathbf{LD}, \mathbf{DE}, \mathcal{D}) \text{ else } \mathbf{0} \,\rangle
\end{aligned}$$

Note that if $\mathbf{M}_1$ is a prototype machine, then $\mathbf{Type}(\mathbf{M}_1) = \mathbf{M}_1$. Here **0** is a degenerate machine with no actions.

This almost gives us the semantics that we want, except for a possible problem in the order of evaluation: we have not constrained the argument machines to ensure that they are evaluated (executed) before the subprogram itself is evaluated. To accomplish this, for each argument that requires elaboration, we introduce a temporary local variable to carry the result of the argument evaluation forward to the subprogram itself. Ignoring type checking again, and leaving out the conditionals (we assume that both arguments require elaboration) we now have:

$$\begin{aligned}
&\mathbf{E}(\llbracket \textit{subprogramName} \texttt{ (<argument}_1\texttt{>, <argument}_2\texttt{>);} \rrbracket, \mathbf{LD}, \mathbf{DE}, \mathcal{D}) \\
&= \langle\, \mathbf{V}_1 \mid \mathbf{V}_2 \mid \mathbf{E}(\llbracket \texttt{<argument}_1\texttt{>} \rrbracket, \mathbf{LD}, \mathbf{DE}, \mathcal{D})[\mathbf{V}_1/\texttt{out}]; \\
&\quad \mathbf{E}(\llbracket \texttt{<argument}_2\texttt{>} \rrbracket, \mathbf{LD}, \mathbf{DE}, \mathcal{D})[\mathbf{V}_2/\texttt{out}]; \\
&\quad \mathbf{New}(\mathbf{Ref}(\textit{subprogramName}, \mathbf{DE}))[\mathbf{V}_1/\mathbf{P}_1, \mathbf{V}_2/\mathbf{P}_2] \,\rangle \backslash \mathit{Sort}(\mathbf{V}_1) \cup \mathit{Sort}(\mathbf{V}_2)
\end{aligned}$$

where $[\mathbf{V}_1/\texttt{out}]$ relabels the output formal parameter with the actions of $\mathbf{V}_1$. Note that we have somewhat arbitrarily determined an order of evaluation for the actual parameter expressions.

In [3] we have explored event more complex type checking and overload resolution schemes. While these schemes add significantly to the type information that is passed around between references (the reference functions themselves become very complicated), the basic structure of the elaboration in terms of the structure of machines still remains the same.

A final note on subprogram calls – we have made no distinction between functions and procedures in this semantics, nor have we made any distinction between calls that occur as an actual statement and calls that occur as part of a subexpression. We note that if one of the actual arguments to the subprogram was accidentally a procedure, then the typed reference to the subprogram would fail to resolve to a machine. This further illustrates the generality of this approach to semantics.

5.5 Declarative Blocks

A declarative block is a collection of declarations followed by a sequence of statements and possibly an exception handler. There are a number of possible semantics for declarative blocks, each reflecting a different visibility semantics for the declarations that occur in the block. We show two possibilities here.

For let or let* visibility semantics[14], we have:

$$\mathbf{E}([\![\ \textit{blockname}\texttt{:}\ \texttt{declare}\ \langle\text{declarative part}\rangle\ \texttt{begin}\ \langle\text{sequence of statements}\rangle\ \texttt{exception}\ \langle\text{exception handler list}\rangle\ \texttt{end}\ ]\!],\mathbf{LD},\mathbf{DE},\mathcal{T})$$

$$= \quad \mathbf{M} \equiv \quad \langle\ \mathbf{E}([\![\ \langle\text{declarative part}\rangle\ ]\!],\mathbf{M{:}LD},\mathbf{DE},\mathcal{T})\ |$$
$$\mathbf{E}([\![\ \langle\text{sequence of statements}\rangle\ ]\!],\mathbf{M{:}LD},\mathbf{M{:}DE},\mathcal{T})\ |$$
$$\mathbf{E}([\![\ \langle\text{exception handler list}\rangle\ ]\!],\mathbf{M{:}LD},\mathbf{M{:}DE},\mathcal{T})\ \rangle$$

Here **M:LD** is a new local declaration set associated with the newly created machine M, and **M:DE** is a new direct environment associated with the same machine.

We have the following side effects:

$$\mathbf{LD} = \mathbf{LD} \cup \{(\textit{blockname}, \mathbf{M})\}$$

$$\mathbf{M{:}DE} = \mathbf{M{:}LD} \underset{m}{\cup} \mathbf{DE}$$

Here $\underset{m}{\cup}$ denotes a modified set union [4][5] that formalizes the hiding of some member of the second set (**DE**) by members of the first set (**M:LD**).

For letrec[15] visibility semantics the direct environment passed to the declarative part would be different, giving:

$$\mathbf{M} \quad \equiv \quad \langle\ \mathbf{E}([\![\ \langle\text{declarative part}\rangle\ ]\!],\mathbf{M{:}LD},\mathbf{M{:}DE},\mathcal{T})\ |$$
$$\mathbf{E}([\![\ \langle\text{sequence of statements}\rangle\ ]\!],\mathbf{M{:}LD},\mathbf{M{:}DE},\mathcal{T})\ |$$
$$\mathbf{E}([\![\ \langle\text{exception handler list}\rangle\ ]\!],\mathbf{M{:}LD},\mathbf{M{:}DE},\mathcal{T})\ \rangle$$

5.1 *Declarative Part*

The elaboration of the declarative part simply elaborates each of the declarations, composing any machines that result in parallel. It is important to note that the elaboration of a variable will return a machine. The elaboration of a function declaration or type declaration *will not return a machine* - the created machine will be associated with the declared name in the local declaration set **LD**, but no machine is actually instantiated as part of the elaboration.

For let or letrec visibility semantics, we would have

14. Let and let* [11] are constructs arising in the language scheme in which declarations in a block are not visible at all to each other (let semantics) or earlier declarations are visible to later declarations (let* semantics). [4][5] cover the variations in visibility semantics in more detail.

15. In letrec semantics [11] declarations in a block are all mutually visible, thus allowing recursive declarations.

$$\mathbf{E}([\![\texttt{<declarative part>}]\!], \mathbf{LD}, \mathbf{DE}, \mathcal{T})$$
$$= \quad \langle \quad \mathbf{E}([\![\texttt{<declaration}_1\texttt{>}]\!], \mathbf{LD}, \mathbf{DE}, \mathcal{T}) \mid$$
$$\mathbf{E}([\![\texttt{<declaration}_2\texttt{>}]\!], \mathbf{LD}, \mathbf{DE}, \mathcal{T}) \mid ... \mid$$
$$\mathbf{E}([\![\texttt{<declaration}_n\texttt{>}]\!], \mathbf{LD}, \mathbf{DE}, \mathcal{T}) \rangle$$

It is important to note that for letrec visibility semantics, the enclosing declarative block has included **LD** in the computation of **DE**. Thus the elaboration of one declaration could well affect the meaning of a reference in another. This points out the importance of the order of elaboration in determining the meaning of a program. Some languages put such severe constraints upon the relative positions of declarations with respect to references that the order of elaboration is not an issue. Other languages, like Ada, provide mechanisms to specify the order of elaboration in cases where the order may not be sufficiently constrained[16]. In [4] we show that an appropriate ordering, if one exists, may be determined through the construction of a dependency graph relating declarations and references. Cycles in this graph indicate that no proper elaboration ordering exists.

For let* visibility semantics, we would have

$$\mathbf{E}([\![\texttt{<declarative part>}]\!], \mathbf{LD}, \mathbf{DE}_0, \mathcal{T})$$
$$= \quad \langle \quad \mathbf{E}([\![\texttt{<declaration}_1\texttt{>}]\!], \mathbf{LD}_1, \mathbf{DE}_0, \mathcal{T}) \mid$$
$$\mathbf{E}([\![\texttt{<declaration}_2\texttt{>}]\!], \mathbf{LD}_2, \mathbf{DE}_1, \mathcal{T}) \mid ... \mid$$
$$\mathbf{E}([\![\texttt{<declaration}_n\texttt{>}]\!], \mathbf{LD}_n, \mathbf{DE}_{n-1}, \mathcal{T}) \rangle$$

Here the order of elaboration is defined to be the order of declaration. For these elaborations, we have:

$$\mathbf{LD}_0 = \varnothing$$

Prior to the ith elaboration, we have:

$$\mathbf{LD}_i = \mathbf{LD}_{i-1}$$

$$\mathbf{DE}_i = \mathbf{LD}_{i-1} \cup \mathbf{DE}_0$$

After the ith elaboration, $\mathbf{LD}_i$ also contains the declaration resulting from the elaboration. After the last elaboration, we compute the returned set of local declarations:

$$\mathbf{LD} = \mathbf{LD}_n$$

5.5.2 *Sequence of Statements*

Because statements may have labels on them and references to them, elaboration order is important here as well. We show the semantics for letrec style visibility[17] (for se-

16. [1] p. 10-11

17. For sequential elaboration order and let* visibility, the computation of the local and direct environments is exactly the same as for the let* visibility of declarations.

quential or let* visibility, the local and direct environments are computed exactly as for the let* declarative items).

$$\mathbf{E}([\![\texttt{<sequence of statements>}]\!], \mathbf{LD}, \mathbf{DE}, \mathcal{T})$$
$$= \quad \langle \quad \mathbf{E}([\![\texttt{<statement}_1\texttt{>}]\!], \mathbf{LD}, \mathbf{DE}, \mathcal{T}) ;$$
$$\mathbf{E}([\![\texttt{<statement}_2\texttt{>}]\!], \mathbf{LD}, \mathbf{DE}, \mathcal{T}) ; ...;$$
$$\mathbf{E}([\![\texttt{<statement}_n\texttt{>}]\!], \mathbf{LD}, \mathbf{DE}, \mathcal{T}) \rangle$$

6.0 Summary

We have given an outline of how the semantics of a programming language can be constructively given in terms of primitive state machines and compositions of state machines. Thus the semantics of a program is given as an abstract state machine whose structure is constructively specified by the program itself. We have shown that the dominant concepts of a programming language are readily understood in terms of three basic semantic concepts: state machines, state types (sets of isomorphic state machines), and generic machines (parameterized specifications of state types.) We have shown that programs, subprograms and data types all have a uniform interpretation as state types. We have described the relationship between the identifiers in the language and the semantic model elements that they correspond to, and in [4] and [5] we have provided a set-theoretic description of the computation of visibility in programming languages.

Constructive semantics is fully abstract in the sense that behavioral equivalence defines equivalence classes of semantic expressions, and these equivalence classes can be taken to be the fully abstract semantics of the expression. We have left two interesting questions open in this area. Is there a normal form for machine algebra expressions that would ease their syntactic comparison? Is behavioral equivalence decidable in the restricted classes of machines used in our semantics? Brookes' work on normal forms of synchronization trees[18] (which underlie HCCS) leads us to believe that for *finite* machines a unique (up to the ordering of terms) normal form exists in HCCS for each equivalence class, but we suspect that the existence of a normal form of machine algebra expressions is precluded by the constraints that our machine algebra places on the form of HCCS expressions.

References

1. Ada Programming Language, ANSI/MIL-STD-1815A (1983)

2. Brookes, Stephen D, A Model for Communicating Sequential Processes, Ph.D. thesis, University College, Oxford University (1983)

18. [2] pp. 99-100.

3. Brown, Paul C., Oconnor, D.M., and Kelliher, Tim, "An Extended Overload Resolution Algorithm that allows Types and Subprograms as First Class Objects," internal document, GE Corporate Research and Development Center, Schenectady, New York (1989)

4. Brown, Paul C., Computing Visibility in Programming Languages, Technical Report 90CRD098, GE Research and Development Center, Schenectady, New York 12301 (1990)

5. Brown, Paul C., Constructive Semantics, Ph.D. thesis, Rensselaer Polytechnic Institute, Troy, New York (1992)

6. Eaker, Charles E., "Creating Software Should Be Easy," (unpublished) GE Corporate Research and Development Center, Schenectady, New York (1991)

7. Eaker, Charles E., "How to Create Software: A Guide to the Perplexed," (unpublished) GE Corporate Research and Development Center, Schenectady, New York (1991)

8. Hoare, C.A.R., Communicating Sequential Processes, Prentice Hall International (1985)

9. Milner, Robin, "Calculi for Synchrony and Asynchrony," Journal of Theoretical Computer Science, Vol. 25, pp 267-310 (1983)

10. Milner, Robin, Communication and Concurrency, Prentice Hall International (1989)

11. Rees, Jonathan and Clinger, William (eds), "Revised3 Report on the Algorithmic Language Scheme," SIGPLAN Notices, Vol. 21, No. 12 (1986).

12. Rumbaugh, James et. al., Object Oriented Modeling and Design, Prentice Hall (1991)

13. Van Benthem, Johan, The Logic of Time: A Model-Theoretic Investigation into the Varieties of Temporal Ontology and Temporal Discourse, D. Reidel Publishing Company (1982),