THE FRAME MODEL OF COMPUTATION

Mitchell Wand
Computer Science Department
Indiana University
Bloomington, Indiana 47401

TECHNICAL REPORT NO. 20

THE FRAME MODEL OF COMPUTATION

MITCHELL WAND

DECEMBER 1, 1974

This report has been submitted for publication elsewhere and has been issued as a Technical Report for early dissemination of its contents.

The Frame Model of Computation

Mitchell Wand

December 1, 1974

Computer Science Department
Indiana University
Lindley Hall 101
Bloomington, Indiana 47401

Key words and phrases: frame model, formal semantics, modular programming, reduction, control regimes, coroutines, backtracking, direct semantics, axiomatic semantics.

CR Categories: 5.24, 4.22

The Frame Model of Computation

Mitchell Wand

Indiana University

Abstract: This paper gives an informal description of a new semantic model of computation called the frame model.

A common criticism of formal semantic models is that they fail to be perspicuous. One may spend so much time on coding details that essential concepts are obscured. An extreme example would be the use of a Turing machine as a semantic model. One reason the Turing machine makes an inadequate model is that its structure is quite different from the structure of programming languages or of "well-structured" computations. It is the purpose of this paper to describe a semantic model of computation whose structure we believe is well-suited to the description of computations. A well-structured model would also be a "frame" in the sense of Minsky [24]: a declarative structure whose components correspond to the terms in which one normally thinks of a computation.

We view a semantic model as describing a class of programming languages which work in essentially the same way, but have different elementary operations. We will indicate two different ways in which the frame model may be turned into a programming language.

Section 1 is concerned with some epistemological vinegar. Section 2 describes the internal workings of the model. Section 3 is essentially a defense of our choice of components. Section 4 returns to the transition from model to language. Section 5 suggests the formal methods underlying our development, and Section 6 compares the frame model with related systems.

1. Semantic Models and Programming Languages

In a traditional semantic model, two steps must be performed before the model can execute an algorithm. The programmer first expresses the algorithm as a program in some programming language.

The program is then transformed into an initial state of the model. In principle, therefore, the set of initial states of a semantic model constitutes a programming language just like any other. In practice, however, no one would write an applications program in the language of initial states of any existing semantic model -- the structure of the models is not well-suited to writing programs. One goal of a perspicuous model, therefore, is to make the associated language of initial states a reasonable programming language.

If our goal is a model with "appropriate structure," it is reasonable to ask what it is we mean by "a structured set." A set, after all, is identical up to renaming (bijection) to any other set of the same cardinality. What is distinctive about a structured set is its accompanying description: " S has 2 elements and comes equipped with an associative binary operation and a two-sided identity for it." One may then refer to the elements of S in terms of this description, e.g., "the identity element" or "the non-identity element." It makes no difference whether the set S is $\{a,b\}$, $\{0,1\}$ or $\{\varepsilon,\gamma\}$. This is nothing more than the "axiomatic approach": rather than specifying a set by naming its elements, (which we call the "exhaustive approach"), we give axioms which the set must satisfy. Then the language in which the axioms are given can also be used to specify elements of the set.

The structure of our semantic model, therefore, carries with it a language -- the language in which the structure is specified. Furthermore, since this is the language in which initial states are specified, it is also something very much like a programming language. In general, these relationships are clearest in the proper mathematical context, which is the theory of categories [23], but it is in part the purpose of this paper to describe the progression from structure to language in a particular case. In order to do that, we must describe our semantic theory.

2. The Frame Model

Our model is based on the "little man" metaphor of Papert [25]. We imagine a community of little men, who converse in order to perform computational tasks. If a particular man M is given a datum on which to work, he might do some work on it and pass the altered datum to M', along with a note saying: "When you have finished with this datum, please pass your answer to M", who will finish the job." (It is, of course, up to M' to decide whether to follow this recommendation.) A typical stage in a computation then consists of a "man" instantiated with a datum to work on. We call such an instance a frame. Most frames consist of the following:

(1) an action which specifies the task to be performed by the frame,

(2) a datum or argument on which the action is to be performed,

(3) a binding which specifies the meanings of identifiers occurring in the action, and

(4) a continuation which specifies the frame to be executed when the action is completed.

Notice that as yet we have made no commitments as to what actions, data, bindings, or continuations look like. We have not, for example, made an *ab initio* distinction between local and global variables or imposed any kind of stack discipline. We will draw such a frame as in Fig. 2.1. (Note that the components are *reversed* from the above discussion; this choice seems to yield more comprehensible diagrams.)

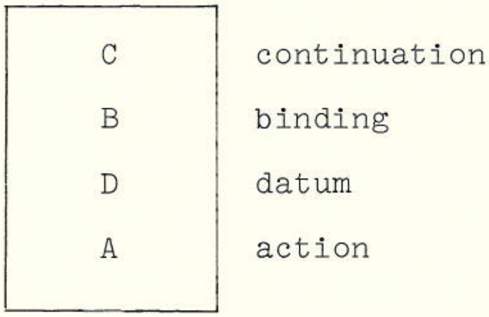

Figure 2.1: Schematic symbol for a frame

Let us now consider how a complex action might be performed. The most obvious nonsimple action is a *composition*. If a1 and a2 are two actions, let us denote by a1;a2 the action which is their serial composition. How would the serial action a1;a2 be performed? Clearly [9] one ought to perform a1 in a frame whose continuation is a frame whose action is a2 . So, to execute the frame F_1 in Fig. 2.2, we execute instead the frame F_2 . We say F_1 *reduces to* F_2 , and write $F_1 \vdash F_2$. We will call F_1 the *antecedent* and F_2 the *consequent*.

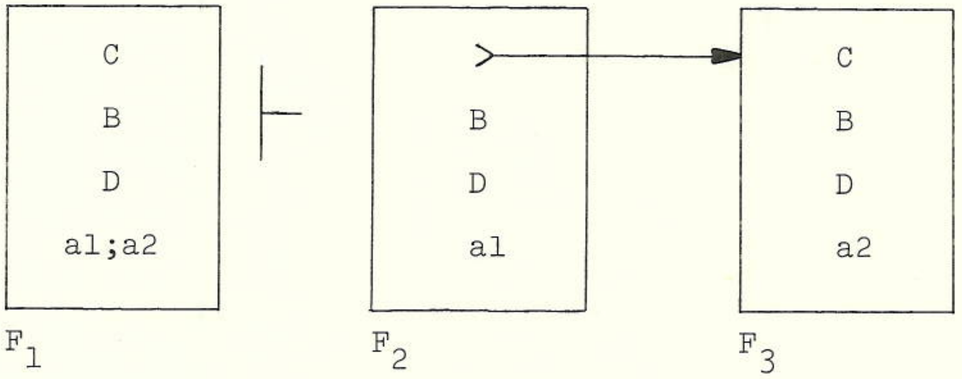

Figure 2.2: Executing a serial action

Note that in Fig. 2.2, F_3 is a part of F_2 . The use of arrows in no way implies that we are talking about pointer variables; it is merely a graphical device to increase legibility.

Another thing a frame might do is to ask another frame to process its value. Such an action is called an invocation. We call the frame which the current frame is trying to invoke the target (this terminology is due to Hewitt [15]). This reduction is shown in Fig. 2.3. Since a frame is to be a complete computational entity, any identifiers appearing in the target's action should refer to the target's bindings; if the situation arose from a serial decomposition then the current frame's bindings are available in the continuation. (See Fig. 2.4.)

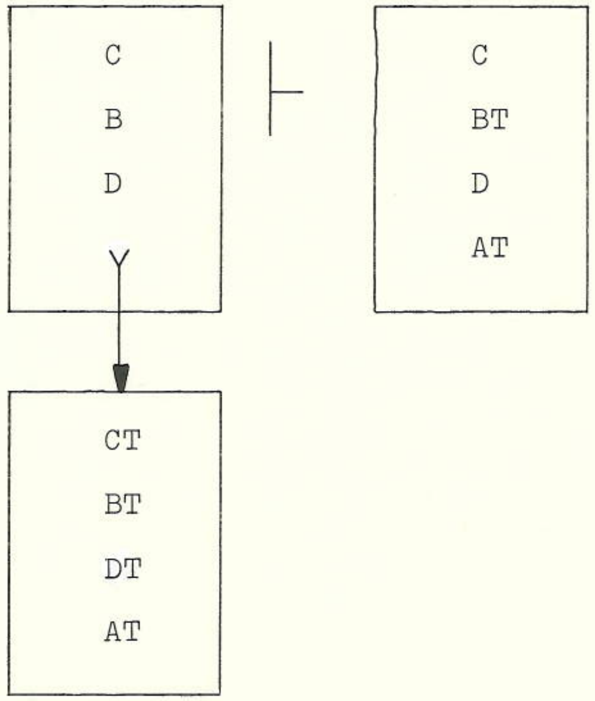

Figure 2.3: Invoking a target frame

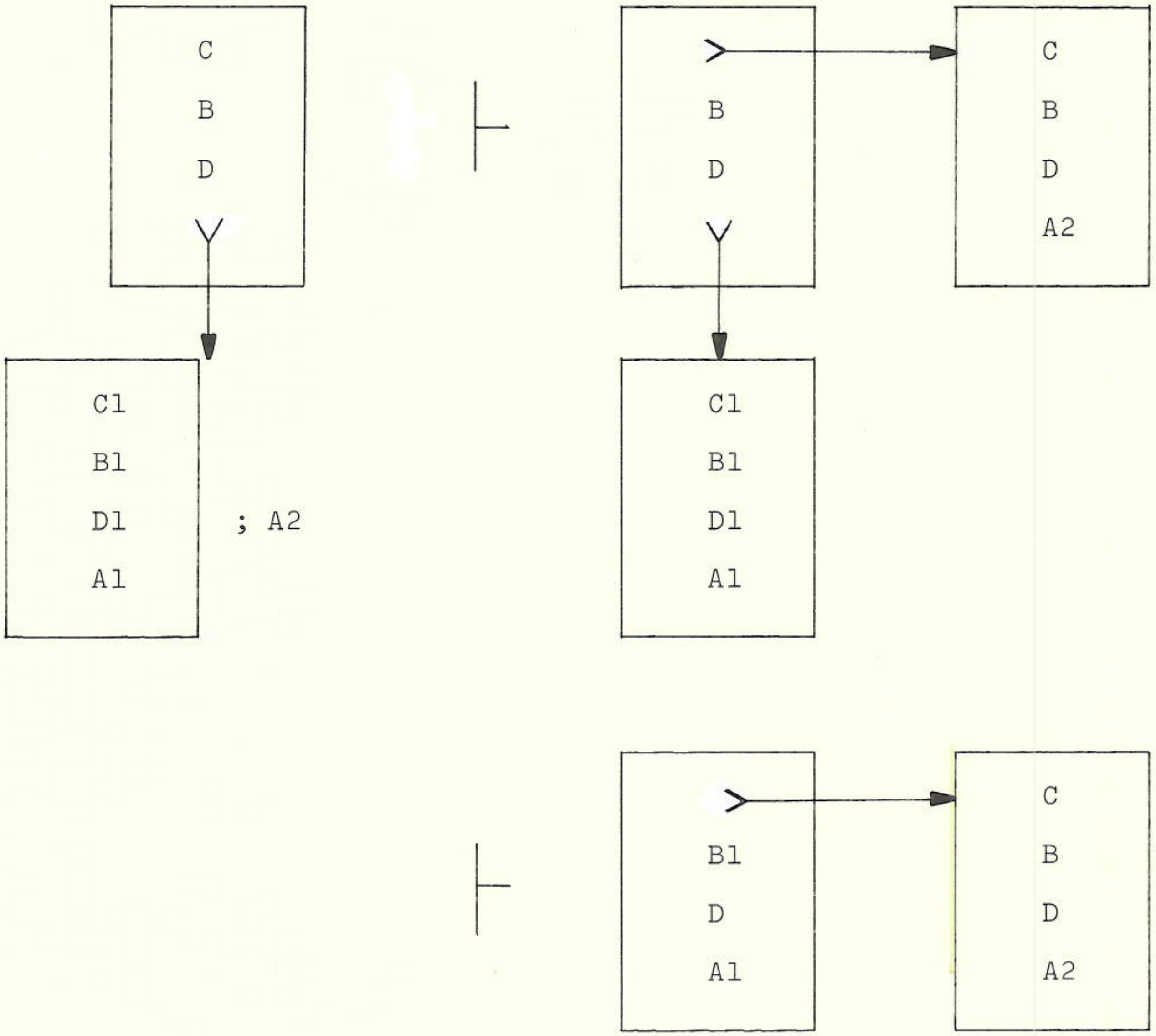

Figure 2.4: A more complex action

Reductions in which one frame attempts to start another occur in many situations. We call such a reduction a *transaction* between the two frames. Transactions will be studied in greater detail in section 3.

We distinguish four types of actions in the model. The first two, composition and invocation, we have already explained. The others are *transmission* and *identification*. An action of transmission is used to perform an action with an argument other than the current datum. Transmission is typically used to call a subroutine with an argument. The reduction for a transmission is shown in Figure 2.5. An identification is used when the target frame is known by name only. This typically occurs in a recursive call or in interpreted code. The reduction for an identification is shown in Fig. 2.6. In Figure 2.7, a subroutine, referred to by name, is called with an argument from a frame. This typical programming situation illustrates all four types of actions.

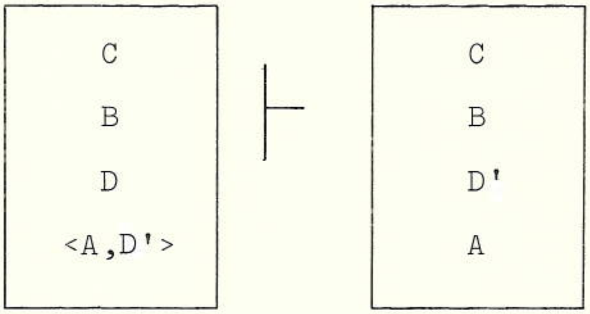

Figure 2.5: Transmission

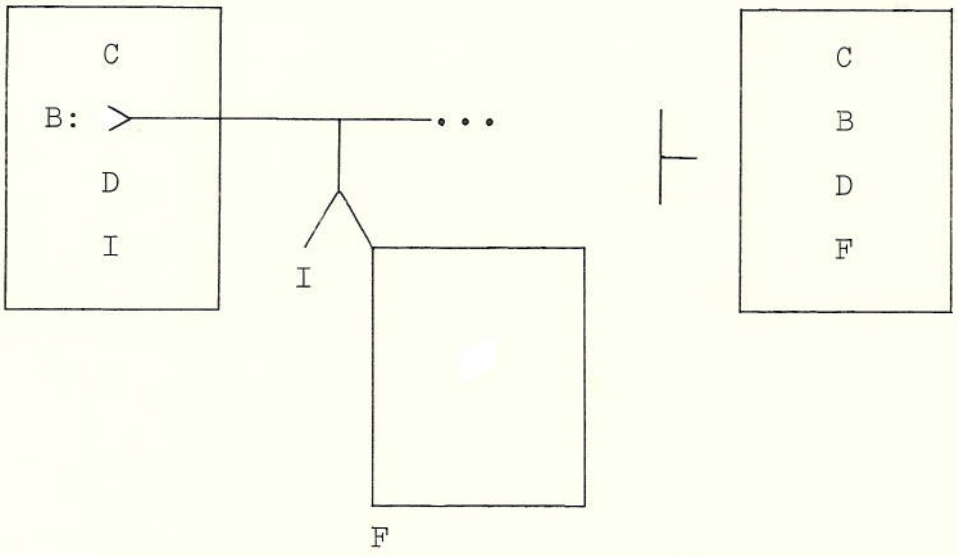

Figure 2.6: Identification

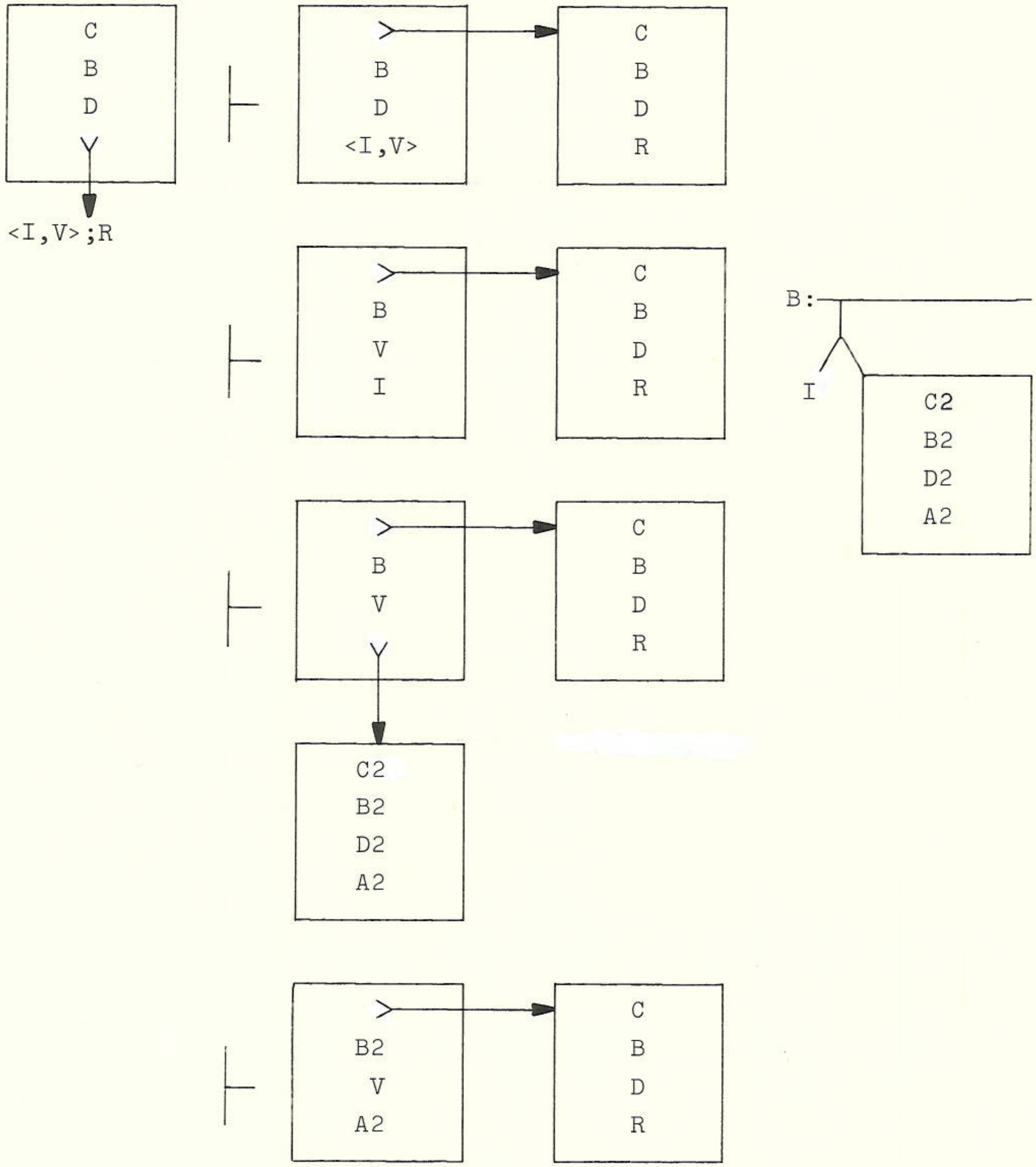

Figure 2.7: Calling subroutine I with argument V

According to this picture, a computation is accomplished by continually applying the four basic reduction rules, breaking down the action of a frame into simpler and simpler subactions. Amidst this vision of frames and actions endlessly dividing and subdividing, one might easily ask: "When does any useful work get done?"

The simple answer is, of course, never. The problem is that there is no provision for any frame to say: "The buck stops here!" It is not sufficient at this point to say, "This only appears to be what happens!" (for an infinite regress appears to terminate only from the point of view of something like generalized computability [27] rather than the more conventional variety) or to appeal to a lattice-theoretical bootstrap argument (the least fixed point of $\lambda x.$"undefined" is still "undefined"). What we have is an induction argument without a base step, or a grammar missing terminal productions, and the solution in either case is simple: add what was missing.

We must, therefore, add something like the notion of an atomic frame to the model. Again it is insufficient to say that some frames will be executed by hardware, for there are at least two quite distinct issues involved:

(1) What frames will the programmer regard as predefined?

(2) What frames will the implementor regard as predefined?

Our answer to the first question determines what features will be available to the programmer, while the second determines how these features will be achieved.

In an exhaustively defined model, it is easy to confuse these two issues: if one says "frame F is atomic" then F is atomic to all concerned. On the other hand, in an axiomatic system, one can say, "There is a frame F which performs task T ." Whether F is atomic, has the usual four components, or is some transcendental beast not explicitly describable in our terms is not specified by this axiom, but the programmer has what he needs to know: what F does. The implementor is then free to implement F as he sees fit.

There are, therefore, two notions which are confused in an exhaustive model: the concept of an atomic frame, which is an implementation concept, and the concept of "a frame F which does task T ," which is a semantic concept. In our model, we express doing task T as a reduction: that is, corresponding to each such frame there will be a set of reduction rules (one such rule is shown in Fig. 2.8). We therefore call these axiomatic frames.

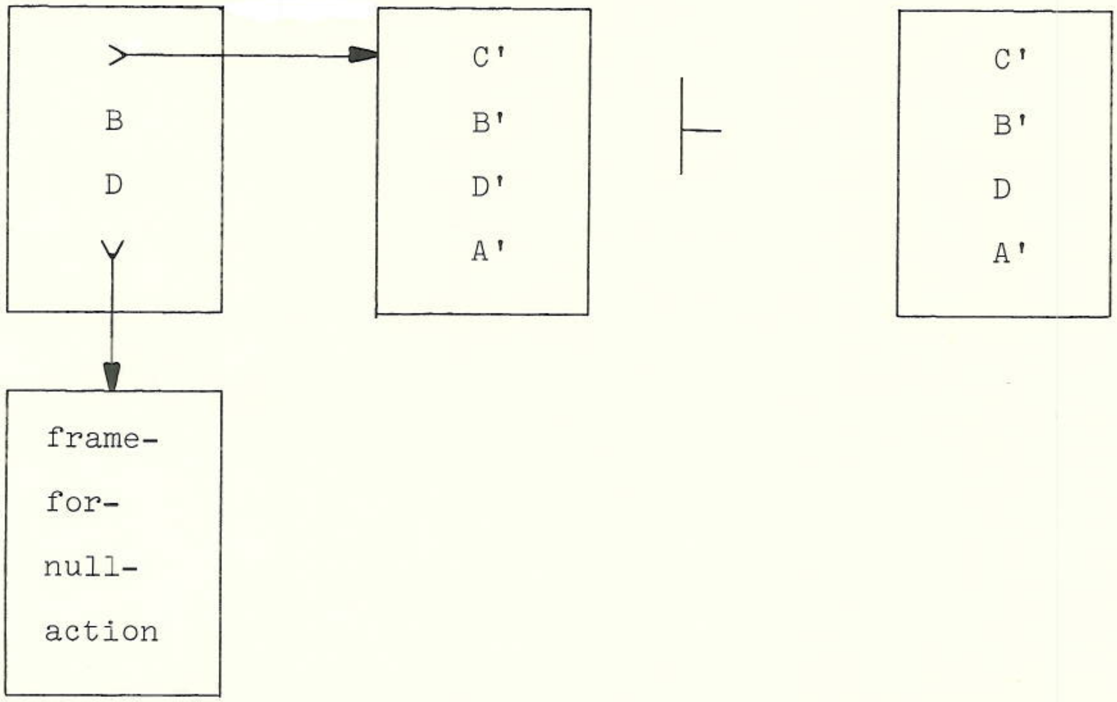

Figure 2.8: Sample reduction rule for axiomatic frame

We are now left with the task of finding a sufficient set of axiomatic frames. Before giving an answer, it will be useful to preview some of the tasks that our atomic frames will be called upon to perform.

3. The Universality of Reduction

In the last section we saw two examples of the notion of reduction: to execute a frame (the antecedent) with a complex action, we created another frame (the consequent) with a simpler action and executed it instead. Reduction is a universal control primitive: all known control flow/data flow regimes are expressible as reductions. In this section we will describe some traditional programming language constructs in terms of reductions in the frame model.

To say that reduction is a universal control primitive is not a strong contention. All it says is that for every nonfinal frame F , there is a frame $\delta(F)$ such that $F \vdash \delta(F)$. In other words, there is a set Q of frames (states of the model) and a "next state" transition map $\delta : Q \to Q$; that is, we have an infinite automaton or information structure model [30]. Our improvement is that δ becomes "well-structured": given the structure of the frame F it is easy to compute $\delta(F)$. Just how easy will be seen from the examples.

1. The Subroutine return. The subroutine is the current frame; the caller to which the subroutine is trying to return is the target. The action to be performed is the action of the target; since this is a return the target's bindings are restored. The subroutine

returns its value by placing it as the datum of the consequent. Last, a returning subroutine assumes that its caller knows how to proceed further, so the continuation of the consequent is the target's continuation. This situation is summarized in Figure 3.1.

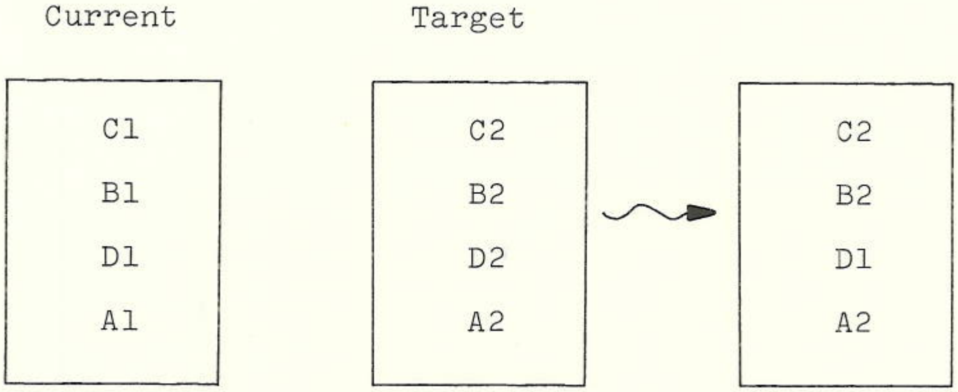

Figure 3.1: Executing a return

Here the current frame is the antecedent and the target its continuation, but this need not always be the case. For example, this transaction also models the resume typical of backtracking systems. There the current frame is the failed frame and the target frame is the "next alternative" frame; everything about the failed frame is forgotten except for the message (datum) it sends to the next alternative. If the current frame were an iterator and the target an iteration operator [7], then the datum returned could include a frame for the consequent to call later on.

2. The subroutine call. The main program is the current frame; the frame corresponding to the subroutine is the target. The action to be performed is the action of the target. Since the subroutine is to be invoked on the datum suggested by the main program, the

datum of the consequent is the datum of the current frame. The bindings of the consequent may be either the bindings of the target (as in ALGOL) or the bindings of the current frame (as in LISP).† The continuation of the consequent should be a frame which performs the remainder of the task of the current frame. This reduction is diagrammed in Figure 3.2.

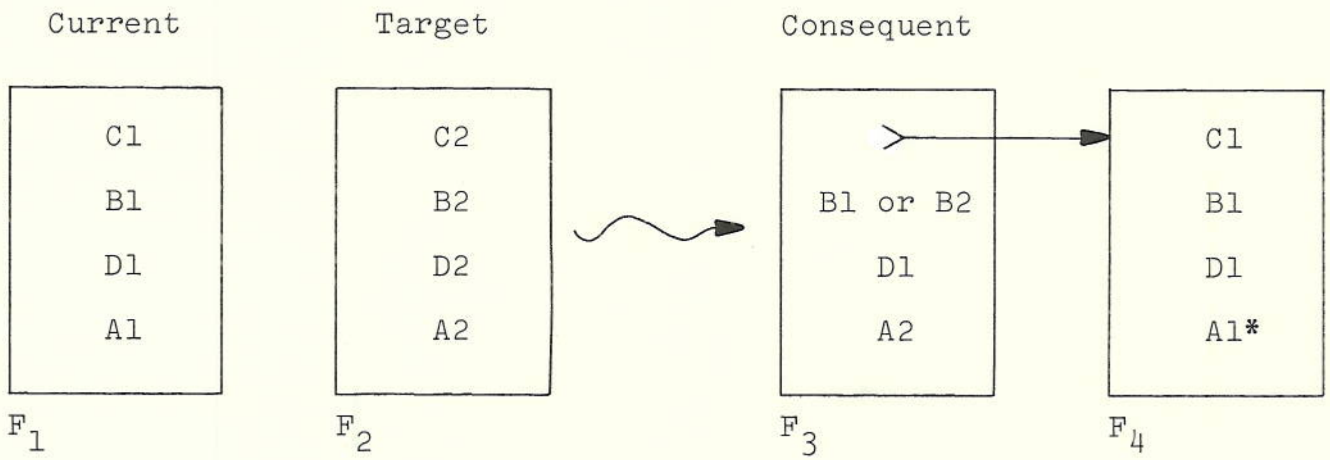

Figure 3.2: Calling a subroutine

Hopefully at some point F_3 will do a return to its continuation, causing F_4 (the remainder of the main program) to be instantiated with the value of the subroutine as its datum, and allowing the main program to continue. One sequence achieving a subroutine-call reduction has already appeared as Figure 2.4.

We summarize the reductions so far in Figure 3.3. Here the entry C or T means the component of the consequent is copied from the corresponding component of the current frame or the target

† These bindings are normally preceived as free variables or non-locals [26]. The first task of the subroutine is to create an appropriate local binding by, say, evaluating its argument (the datum). We view this as the task of the subroutine rather than the interpreter. Typically a utility function is available for argument evaluation.

frame, respectively; the entry C* indicates that a copy of a variant of the entire current frame is inserted in the position. In every case the action of the consequent is the action of the target and the datum of the consequent is the datum of the current frame; this is in keeping with the picture of the current frame trying to "pass the buck" to the target frame. At this point enumerative creativity becomes appropriate: what other combinations are possible?

	Continuation	Binding	Datum	Action	Use
1.	T	T	C	T	return/resume
2.	C*	T	C	T	call(ALGOL)
3.	C*	C	C	T	call(LISP)
4.	T	C	C	T	?

Figure 3.3: Table of transactions

	Continuation	Binding	Datum	Action	Use
4.	T	C	C	T	goto
5.	C	T	C	T	invocation (closed)
6.	C	C	C	T	invocation (open)/macro expansion/structured goto
7.	T	C	T	T	pure side-effect
8.	T	T	T	T	pure restart
9.	C	C	T	T	valueless call (open)
10.	C	T	T	T	valueless call (closed)

Figure 3.4: More transactions

For example, what are we to make of the fourth line in Figure 3.3? The current frame says to the target, "Go whither thou willt goest, but do it with my variables!" This, too, is recognizable as a familiar programming construct: the goto!

3. Invocation of a frame. As we have seen in Figure 2.4, the transaction of calling a subroutine (as in Figure 3.2) may be divided into two steps: a serial decomposition (Figure 2.2) and the invocation of the subroutine's frame (Figure 2.3). The latter is a transaction, so we add it as line 5 of Figure 3.4. In this case, the current frame is the antecedent frame and the target is the antecedent's action. Just as in the case of the return, the same transaction may also occur in other situations. For example, the action of the antecedent may refer to a frame not explicitly but by an identifier (as is customary in interpreted languages). In this case the target is found by referring to the current frame's bindings.

4. Open invocation. The invocation transaction just discussed was one step in the ALGOL-style call. A similar invocation occurs in the LISP-style call. Since the ALGOL-style invocation corresponds to a lambda-calculus closure (an action with all free variables bound), we distinguish this invocation (and the associated call) by calling it open. Another occurrence of this transaction is as a macro expansion: a piece of code (the action of the target) to be inserted in the current frame. Here most often the target is referred to by an identifier in the action of the current frame. This transaction also models the structured goto of Knuth [20] or the "event" of [31], in which a piece of code, referred to by

name, is executed without modifying the stack (continuation).

5. Valueless transfers. Programming languages ever since FORTRAN have distinguished between subalgorithms which return a value and those that do not. In the frame model, these are modeled by the transactions of lines 7 and 8 of Figure 3.4. Note that these are distinct from so-called "null-valued" procedures, which return a value from a data type consisting of a single element. Similarly, one may propose "valueless calls" (lines 9 and 10).

Many other programming language constructs are modellable by reductions as simple as the ones considered here. A frame may cause a "side effect" by modifying the bindings of its continuation prior to resuming it. The action of a subroutine typically starts with a frame which creates the appropriate bindings for the remainder of the action. (For example, it may evaluate its arguments.) This is nothing more than an execution-time declaration. Some of the information in a declaration may be processed prior to execution time; for example, identifiers occurring in actions may be replaced by the frames to which they refer. This may be done either at compile time or at load time (see, for example, [6] on the possible subtleties). Compiled code typically insists that all such "external references" be "resolved" prior to execution, while interpreted code generally relies on identifiers persisting through execution. The frame model allows a systematic study of such trade-offs.

4. Criteria for Axiomatic Frames

In section 3 we suggested that the "transition function" δ should be "well-structured." After reviewing the examples there, we can propose at least one criterion for well-structuredness: every reduction is expressible as a local transformation of the top-level frame. The predefined reductions of section 2 were well-structured, as were all of the transactions of section 3. This suggests a necessity criterion for axiomatic frames: The effect of every axiomatic frame must be describable as a local transformation of the top-level frame. This means that axiomatic frames are describable in the same language (the language of finite diagrams) as the interpreter, and therefore the language is complete [1].

This necessity criterion tells us what axiomatic frames are permissable, but it does not tell us which of the permissable axiomatic frames we should supply. On the other hand, the criterion for well-structuredness does give us a clue. If every reduction is expressible as a local transformation of the top-level frame, then if we capture all the (infinitely many) local transformations, then we surely capture all the reductions. Therefore, we can state a sufficiency criterion: A set of axiomatic frames is adequate if it allows the programmer to express every local transformation.

This says that every transaction of every control structure is equivalent (in a fairly transparent way) to a local rearrangement of a particular data structure.

There are many ways to choose an adequate set of axiomatic frames. Each such set provides a set of initial states in which one can write useful programs -- that is, a programming language. Furthermore, all such languages "work the same way" -- they all rely on the same four basic reductions corresponding to the same four kinds of actions. This explains the comment in the introduction that a semantic model describes a class of programming languages.

One way to obtain an adequate set of axiomatic frames is to notice that the structure of frames is basically a product type so that one may quickly write down one sufficient set of axiomatic frames:

a. Constructors and selectors. For frames, we will have the four obvious selectors and a constructor consframe. Our informal syntax will be <consframe an-action a-datum a-binding a-continuation> . This syntax passes over the need for a convention for multiple-argument transmissions and over the distinction between the identifier consframe and the frame to which it is bound, neither of which are a concern at this point.

b. Frames for evaluating arguments. Constructors and selectors evaluate their arguments, which means that they are not immediately describable in terms of local transformations. This may be solved by introducing some axiomatic frames that evaluate lists of arguments.

c. Predicates. Examination of the formal language of diagrams suggests that our predicates fall into two classes. The first is the class of "form-checkers" such as atom in LISP. The second is the equality predicate. An analysis of the axioms in the formal definition shows that it is sufficient to test equality only on identifiers. Luckily, this axiomatic frame is describable directly in terms of finite diagrams, while a more general equality operator would be much more difficult to formalize.

d. Self-referential operators. This requirement seems well-known to designers of real programming languages (CONNIVER, B&W), but seems to have first been stated clearly in the theoretical literature by Backus [1]. The problem is that a frame needs to know where it is. In CONNIVER, for example, this task is accomplished by the function FRAME, and in RED's by "meta" operations. In the frame model, for a frame to accomplish a reduction, it needs to obtain as a datum the current frame in order to decompose it. We therefore need a self-referential atomic frame. There are various ways to perform self-reference; one choice, called callthis, is diagrammed in Figure 4.1. An action of the form <callthis I> is analogous to (SETQ I (FRAME)) in CONNIVER.

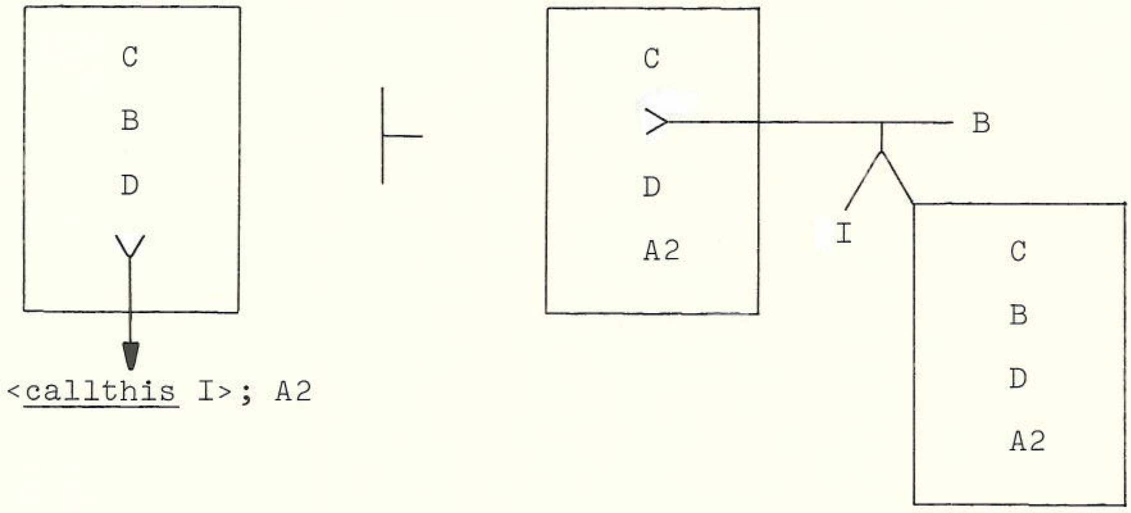

Figure 4.1: Self-reference

e. Starting a new frame. Once having constructed a frame, we have to get it started. This is easily accomplished by the frame startframe, which not only constructs a frame with consframe but also makes the resulting frame the consequent.

Figure 4.2 shows the intended reduction for a simple SETQ-like operation, and Figure 4.3 gives the action for a frame that performs the operation.

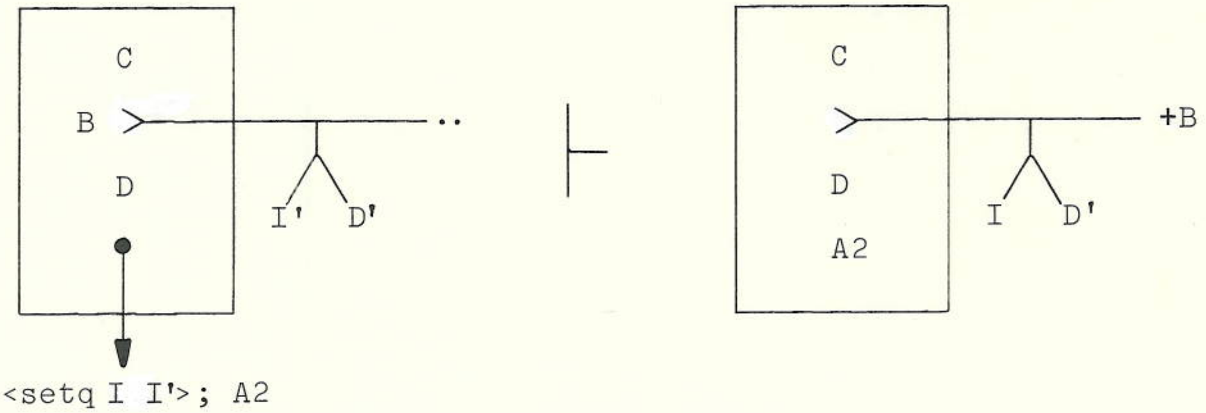

Figure 4.2: Goal for an assignment operator

```
<callthis x>;<startframe<action<continuation x>>
                        <datum <continuation x>>
                        <addbinding  <first<datum x>>
                                     <eval <second <datum x>>>
                                           <binding<continuation x>>>
                        <continuation<continuation x>>>
```

Figure 4.3: Action for setq

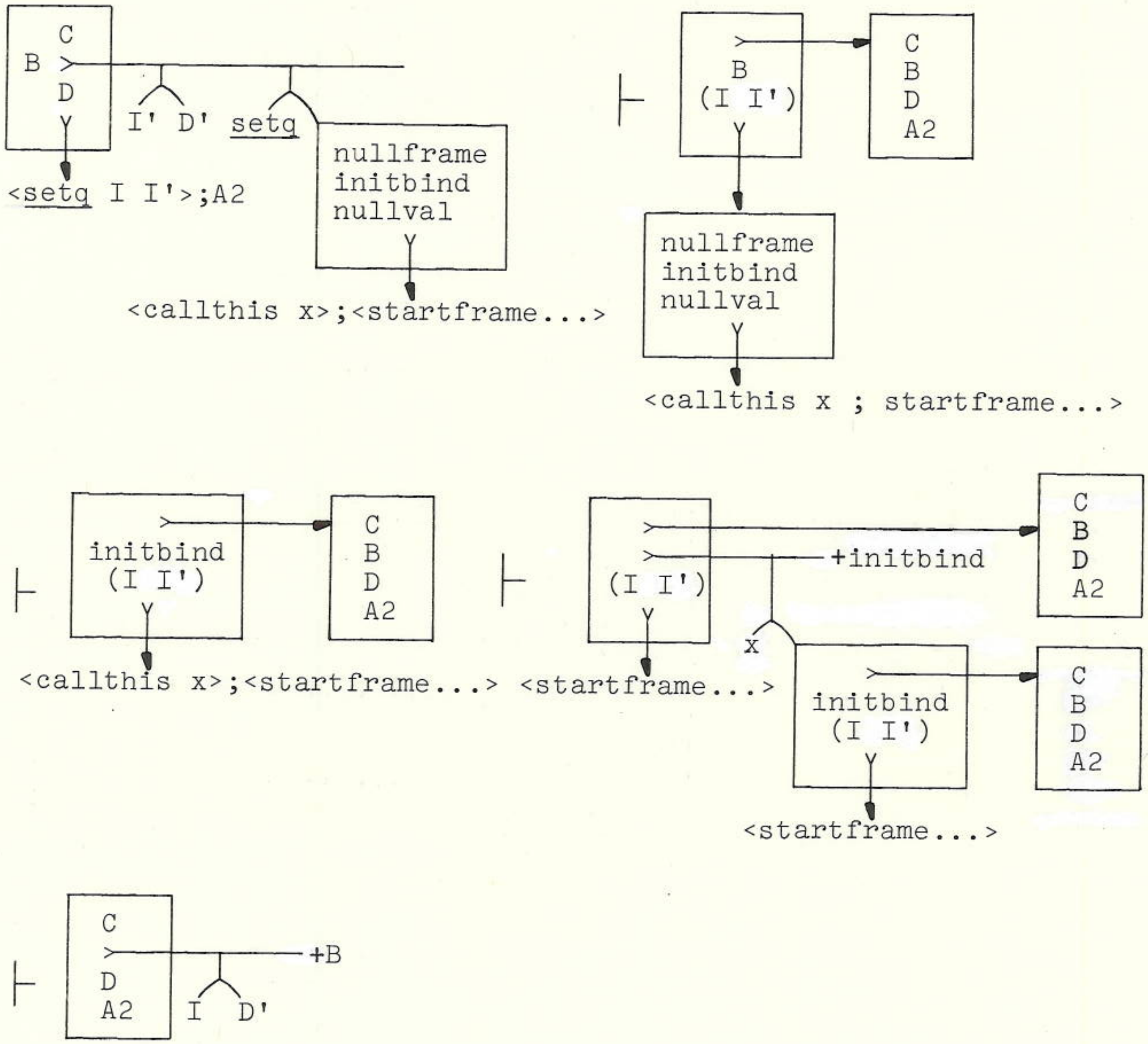

Figure 4.4: Using self-reference with setq.

Figure 4.4 shows some of the reductions undertaken by the setq frame. Clearly any local rearrangement can be coded in similar fashion. This establishes the sufficiency of this set of axiomatic frames.

This is not the only adequate set of axiomatic frames. For example, a somewhat more parsimonious set may be obtained by regarding pattern matching as a primitive operation. The choice of axiomatic frames is very much a choice of programming style.

Of course, to make any of these languages truly practical, there are still many details to be ironed out -- an initial frame for error handling, an initial binding in which some atomic frames are bound to identifiers for use by programmers, and, of course, some useful data like integers and characters--but with the choice of axiomatic frames the semantic portion of the design task is essentially finished.

5. Formal Definition Systems

Our discussion of the frame model has largely been in terms of diagrams. If we seek a formal definition of the frame model, there are several ways in which we may proceed. Although we do not propose to give any formal definition here, it is worthwhile to discuss some of the alternatives.

1. The obvious way in which to proceed is to give a rigorous definition of the set of diagrams and of the relation "$\vdash$" . This makes the set of diagrams a set of machine states and the relation "$\vdash$" a (possibly nondeterministic) transition function. Under such a view, this becomes what is known as an interpretive semantics. We also classify this scheme as an exhaustive semantics, since it attempts to specify a particular set of states and a particular transitive function. This is the niche into which the most popular semantic models fall. The frame model enjoys the advantage over

others in this category of having what we called a "well-structured" transition function.

There are two classes of objections to this style of semantics. The first objection is to the attempt to be exhaustive. We have already suggested two of the pitfalls in exhaustiveness. It is difficult, we argued in section 1, to be simultaneously exhaustive and well-structured. Exhaustiveness also confuses the issue of atomic frames. As we have intimated, we may replace exhuastive systems with axiomatic ones and avoid most of these difficulties. There has been some work on axiomatic interpretive semantics, notably [5].

A second objection is that interpretive semantics lays too much stress on how a computation works and not enough stress on the result of the computation. According to this view, rather than dealing with an infinite automaton (Q,δ) , we should be dealing with a model (L,V,μ) consisting of a set L of programs, a set V of values, and a function $\mu:L \rightarrow V$ (possibly partial or multivalued) mapping a program to its result. This formulation is a variant of that of Knuth [8,18,19], who called it "declarative" semantics; supporters of automaton models call this a compiler-oriented semantics, charging that all this scheme accomplishes is a compilation from one language into another. The viability of this scheme has been demonstrated only very recently, most notably in the work of Scott et al. [28,29] (under the name of "mathematical" semantics) and Backus. In the face of this display of synonymy, we adopt the somewhat more value-free (no pun intended!) appellation "direct semantics" for the (L,V,μ) approach and its variations.

Both Scott and Backus deal with exhaustive direct semantics, but axiomatic direct semantics is also possible, and is perhaps the most attractive of the four possibilities. The most straightforward approach to axiomatic direct semantics is what we might call the "first-order structure" approach. Conventionally, a first-order structure S consists of a set and some functions, predicates, and constants. If Δ is a class of formulas (in, say, first-order logic), we say S is a model of Δ if every formula of Δ is true in S . We may think of Δ as describing the class $M(\Delta)$ of models of Δ . If the formulas in Δ are restricted to certain forms, there are structure theorems that tell us about the class $M(\Delta)$.

To apply this method to our case, we need to introduce the notion of a many-typed structure. Instead of having a single set, our structures will have several sets, one for each "type." In our case, we would have sets of frames, bindings, data, and actions. We would also have functions, predicates, and constants. A diagram becomes a term in the language for such structures. For example, the consequent of figure 2.2 is represented by the term

$$\mathrm{frame(frame(C,B,D,A2),B,D,A1)}$$

where C is a frame-valued variable, B a binding-valued variable, D a datum-valued variable, and A1 and A2 action-valued variables. In addition to these "constructor" functions we introduce a function $\mu:\underline{\mathrm{frames}} \to \underline{\mathrm{data}}$. Now, for each pair of terms $\tau_1 \vdash \tau_2$ in the definition, we introduce the universal closure of the formula

$\mu(\tau_1) = \mu(\tau_2)$ as an axiom. Then an implementation of the frame model is any model of the desctiption Δ .

As it turns out, this procedure is needlessly general, since all the formulas in Δ are of a very special form -- algebraic identities -- and there is a body of mathematical techniques for handling such identities -- categorical algebra, especially the algebraic theories of Lawvere [21] and their extension to many-typed theories [2,10]. In this system, the statement $\tau_1 \vdash \tau_2$ becomes the axiom $\mu \circ \tau_1 = \mu \circ \tau_2$, which asserts the equality of a single pair of maps, rather than the equality of a large class of pairs of elements as does the previous version. Furthermore, the structure theorems give a particularly good picture of $M(\Delta)$, telling precisely how to go about building a model of Δ , and in particular how to build a model and an interpretive semantics with the property that if a computation gives a certain answer in the model, it will give the same answer in every model (as it turns out, almost every model of the frame model has this property). For these reasons, categorical algebra will be used for the formal definition of the frame model.

6. Positive and Normative Models

The last section discussed the technical alternatives for formal definition schemes. In this section we will discuss some of the design decisions in the frame model and compare them with similar decisions in existing semantic models.

It was our intention to have the fewest reasonable number of basic structures. On the other hand, if two things are habitually treated differently, the model should not lump them together. Thus, even though the continuation and the datum could be regarded as parts of the binding, they are made separate because most transactions treat them specially. Similarly, bindings are not frames. To this extent the frame model is a positive model -- that is, it tries to describe how computations really work.

In other respects the frame model is a normative model -- that is, it tries to describe how computations ought to work. In this respect it is somewhat rigid in rejecting distinctions which are judged to be implementation issues rather than semantic issues. In this class we place the distinction between local and nonlocal variables. The semantic issue for any frame is the entirety of its bindings. If a frame wants to change another frame's bindings, it must construct a new frame with the appropriate contents. We have resisted the temptation to make what we think are unnatural acts illegal -- we have merely made them expensive.

We can now discuss the relation of the frame model to four related systems, two more normative than ours and two more positive. The model to which our intellectual debt is greatest is the important PLANNER73 or ACTOR model of Hewitt [11,15,16]. It is also explicitly based on the "little man" metaphor. Its communication is framed in terms of a "message sending" metaphor which is then implemented in terms of reduction: "conceptually at least a new actor is created every time a message is sent" [16, p. 155]. The PLANNER73 model has been criticized for being overly monotheistic [3]; the frame model, because it aspires to positivity, makes some distinctions which we believe are useful. In particular, the frame model demonstrates that the useful control flow/data flow ideas in PLANNER73 can be decoupled from its extensional (or "implicit") data structures. Another feature of PLANNER73 which has caused some confusion is the apparent lack of primitive actors. We believe that our discussion of axiomatic and atomic frames will clarify this issue.

Another model, to which our debt is not so immediately obvious, is the closed applicative language (CAL) model of Backus [1]. The intent of this model was to obtain a rigorously defined class of languages by including only a very small set of components. Such a model is almost entirely normative. The frame model may be thought of as extending the CAL model to make it more positive. The frame model has adopted several important ideas from the CAL model, most importantly the reduction metaphor (and the associated trick for simulating interpretive semantics in a direct semantics). Also from this model comes the notion of a class of languages that behaves the same way (which we identify with the notion of model) and the importance of self-reference.

A third model which resembles the frame model is the GLOSS model of Herriott [12,13,14]. It shares our concern for perspicuity and our emphasis on finite transformation of diagrams. On the basis of the published material, GLOSS appears to be very much more positive and less normative than the frame model.

The frame model, the GLOSS model, and PLANNER73 are all (in lesser or greater measure) generalizations of the contour model [17]. The contour model established the importance of finite diagrams as a mode of explication, and gave a clear discussion of the creation of bindings from local and nonlocal segments. The frame model modifies the contour model by making it more "democratic": the contour discipline on bindings is eliminated, and binding, datum, and continuation are separate and coequal. This choice arises out of our more nomative intentions.

7. Conclusions

We have presented a semantic model of computation called the frame model. We believe that the model is well-suited for discussing semantic issues in programming languages, such as compilation versus interpretation, the processing of declarations, and alternative schemes for coroutining. The frame model also clarifies the relationship between models and programming languages. It emphasizes the importance of the axiomatic approach and supplies a criterion for axiomatic frames. We believe the frame model will be useful not only for the study of semantics of particular programming languages but also for the creation of new models and languages in the future.

Typed by Christopher Charles

References

1. Backus, J. Programming language semantics and closed applicative languages. Proc. 1st ACM Symp. on Principles of Programming Languages, Boston, 1973, pp. 71-86.

2. Benabou, J. Structures algebriques dans les categories. Cahiers de Topologie et Geometrie Differentielle 10 (1973), 1-126.

3. Bobrow, D.G., and Raphael, B. New programming languages for artificial intelligence research. Computing Surveys 6 (1974), 155-174.

4. Bobrow, D.G., and Wegbreit, B. A model and stack implementation of multiple environments. Comm. ACM 16 (1973), 591-602.

5. Burstall, R.M. Formal description of program structure and semantics in first-order logic. Machine Intelligence 5 (Meltzer & Michie, Eds.), Edinburg University Press, 1970, pp. 79-98.

6. Earley, J. Naming Structure and Modularity in Programming Languages, University of California at Berkeley, Technical Report No. 17.

7. ---------. High level operations in automatic programming. SIGPLAN Notices 9, 4 (1974), 34-42.

8. Fang, I. FOLDS, A Declarative Formal Language Definition System, Stanford University Computer Science Report CS-72-329, 1972.

9. Friedman, D.P. The Little LISPer, Science Research Associates, 1974.

10. Goguen, J.A., and Thatcher, J.W. Initial algebra semantics. Proc. 15th IEEE Conference on Switching and Automata Th., New Orleans, 1974.

11. Greif, I., and Hewitt, C. Actor semantics of PLANNER-73. Proc. 2nd ACM Symp. on Principles of Programming Languages, Palo Alto, 1975.

12. Herriot, Robert G. GLOSS: a semantic model of programming languages. SIGPLAN-SIGOPS Interface Meeting, Savannah, Georgia, April 1973, SIGPLAN Notices 8, 9 (September, 1973).

13. ----------------. GLOSS: a high level machine. ACM SIGPLAN-SIGARCH Symposium on the High-level-language Computer Architecture. University of Maryland, College Park, MD, November 1973, SIGPLAN Notices 8, 11 (November, 1973).

14. ----------------. A uniform view of control structure in programming languages. Proc. IFIP 74, 331-335.

15. Hewitt, C.; Bishop, P.; and Steiger, R. A universal modular actor formalism for artificial intelligence. Proc. IJCAI 3, San Francisco.

16. Hewitt, C., et al. Actor induction and meta-evaluation. Proc. 1st ACM Symp. on Principles of Programming Languages, Boston, 1973, 153-168.

17. Johnston, John B. The contour model of block structured processes. SIGPLAN Notices 6, 2 (February, 1971).

18. Knuth, D.E. Semantics of context-free languages. Math Sys. Th. 2 (1968), 127-245.

19. ----------. Examples of formal semantics. Symp. on Semantics of Algorithmic Languages, (E. Engeler, Ed.), Berlin, Springer Verlag, 1971, pp. 212-235.

20. ----------. Structured programming with GOTO statements. Computing Surveys 6 (1974).

21. Lawvere, F.W. Functorial semantics of algebraic theories. Proc. NAS USA 50 (1963), 869-872.

22. McDermott, D.V., and Sussman, G.J. The CONNIVER Reference Manual. MIT Artificial Intelligence Memo No. 259A, 1974.

23. MacLane, S. Categories for the Working Mathematician, New York, Springer-Verlag, 1971.

24. Minsky, M. A Framework for Representing Knowledge. MIT Artificial Intelligence Memo No. 306, June, 1974.

25. Papert, S.A. Teaching children to be mathematicians versus teaching about mathematics. Int. J. Math. Educ. Sci. Technol. 3 (1972), 249-262.

26. Pratt, T. Programming Languages (to appear).

27. Rogers, H. Theory of Recursive Functions and Effective Computability, New York, McGraw-Hill, 1967.

28. Scott, D. Outline of a mathematical theory of computation. Proc. 4th Ann. Princeton Conf. on Info. Sci. & Sys., 1970, 169-176.

29. Scott, D., and Strachey, C. Toward a mathematical semantics for computer languages. Computers and Automata, J. Fox (Ed.), New York, Wiley, 1972, 19-46.

30. Wegner, P. Operational semantics of programming languages. Proc. ACM Conf. on Proving Assertions about Programs, Las Cruces, 1972, 128-141.

31. Zahn, C.T. A control statement for natural top-down structured programming. Symp. on Programming Languages, Paris, 1974.