Fen—An Axiomatic Basis for Program Semantics

B.J. MacLennan
Florida State University

A formal system is presented which abstracts the notions of data item, function, and relation. It is argued that the system is more suitable than set theory (or its derivatives) for the concise and accurate description of program semantics. It is shown how the system can be used to build composite data types out of simpler ones with the operations of rowing, structuring, and uniting. It is also demonstrated that completely new primitive types can be introduced into languages through the mechanism of singleton data types.

Both deterministic and nondeterministic functions are shown to be definable in the system. It is described how the local environment can be modeled as a data item and how imperative statements can be considered functions on the environment. The nature of recursive functions is briefly discussed, and a technique is presented by which they can be introduced into the system. The technique is contrasted with the use of the paradoxical combinator, *Y*. The questions of local and global environments and of various modes of function calling and parameter passing are touched upon.

The theory is applied to the proof of several elementary theorems concerning the semantics of the assignment, conditional, and iterative statements.

An appendix is included which presents in detail the formal system governing *webs* and *fen*, the abstractions used informally in the body of the paper.

Key Words and Phrases: semantics, formal systems, lambda-calculus, extensible languages, data types, modes, axioms, correctness, formal language definition, formal description, data structures, description languages, models of computation

CR Categories: 4.22, 5.21, 5.24, 5.26

1. Introduction

Axiomatic set theory provides a universal basis for mathematical discourse. By this we mean that all mathematical concepts can be expressed in set theory and all proofs are ultimately proofs in set theory. Similarly, set theory can serve as a basis for the study of program semantics; indeed, most current approaches to this topic in some sense use set theory. This paper develops an axiomatic theory which is felt to provide a more intuitive system in which to describe program semantics. Although the system is coextensive with set theory, its basic axioms deal with functions and structured data rather than classes. It is believed that such a system will facilitate programming language definition and assertion proving.

2. Webs and Fen

Structured data items are usually placed in two categories: either they are *primitive*, i.e. in some sense indivisible, or they are *composite*, i.e composed of other composite or primitive data items. It will be shown later that this division is to a great degree illusory, the distinction being based on program efficiency considerations. With this in mind, we will concentrate on composite data items in the following discussion.

Composite data items are characterized by being composed of some finite number of *fields*, which are themselves data items. Each field of a composite data item has an associated *selector*, which is also a data item. Given a composite data item and a selector we can find the corresponding field, a process known as *selection*. If x is a selector of a data item d and if y is a field selected by this selector, then this fact is represented by $d[x{:}y]$.

As an aid to visualization, composite data items can be considered nodes in a directed graph. Interpreted in this way, $d[x{:}y]$ means that an edge labeled x is directed from node d to node y.

The notation $[x_1{:}y_1, x_2{:}y_2, \dots, x_n{:}y_n]$ is used to represent any composite data item with selectors $x_1, x_2, \dots, x_n$ and fields $y_1, y_2, \dots, y_n$ in which x_i selects y_i. The class of all such data items is represented by

$\{x_1{:}y_1, x_2{:}y_2, \dots, x_n{:}y_n\}$

and is called the *performing class* of the items. The distinction between individual data items and the class of all similar data items is one that is difficult to make in systems based on set theory. The techniques used in these systems to avoid this problem are exemplified by

Author's present address; Computer Science Department, Purdue University, Lafayette, IN 47906.

Bekić and Walk [1] who require all values to have distinct names. It is shown later that the above distinction is crucial to the discussion of program semantics.

2.1. Fen and Intentional Data Items

A *fen* (functional entity) is a composite data item all of whose selectors are distinct. If f is a fen and x is any data item, then there is at most one y such that $f[x{:}y]$. Since this y is unique (if it exists), it can be represented by $f[x]$ or, when unambiguous, by[1] fx. Considered in this way, fen are seen to be analogous to finite partial functions on the domain of data items. For example, the fen $[a{:}0, b{:}1]$ corresponds to the partial function F on the domain $\{a, b\}$ which satisfies $F(a) \equiv 0$ and $F(b) = 1$. It can be shown that there is an isomorphism between performing classes and finite partial functions on data items.

The close relation between fen and functions suggests an obvious generalization, viz. the specification of data items by *intension*. If for some function on data items, F, we have that the data item w is represented by $[x \mid F(x)]$, then we have that $w[x] \equiv F(x)$.[2] If the use of braces to denote performing classes is extended to include intensional items, then it can be seen that there is a correspondence between the fen class $\{x \mid M\}$ and the lambda expression $\lambda x\{M\}$, see [4, 5].

The special case where F is a Boolean function deserves some note. Suppose s is the data item $[x \mid P(x)]$, for some predicate P on data items. If S is the set coextensive with P then

$$x \in S \leftrightarrow P(x) \leftrightarrow s[x].$$

In addition, there is an isomorphism between sets of data items and performing classes of the form $\{x \mid P(x)\}$, where P is a predicate on data items. The technique of representing sets as functions onto {**true, false**} is exactly analogous to that used by von Neumann in his axiomatic set theory [22, 23], and is one way to represent infinite sets in finite memory.

2.2. Determinism

No assumption has been made that requires the selectors of a data item to be distinct. Although this possibility may seem counter to our intuitive conception of structured data, it is in fact necessary for a complete treatment of program semantics. Many normal, well defined programs in high level languages will embody nondeterministic states in their execution. Consider the case of storage allocation. When a program requests the allocation of storage for a **real** value, it is only derivable that the pointer returned will refer to an otherwise unused cell of shape **real**. It is not (and for implementation efficiency should not be) derivable what cell is allocated. As another example, consider the parallel execution of $x := 0$ and $x := 1$. The execution of this pair of statements is nondeterministic, but the results are not completely undefined. In particular, we should be able to derive that either $x = 0$ or $x = 1$. Indeed, the program containing this pair might well be deterministic if it computed the same function for $x = 0$ and $x = 1$. Controlled nondeterminacy of this type is a frequent occurrence in high level language systems and must be describable in any system intended to represent programming language semantics. Several other systems [3, 20] do include the concept of an undefined quantity but do not attempt to specify what set of values the undefined quantity represents in a given situation. It is claimed that such a facility is necessary to accurately portray the actions of reasonable programs. In particular, it is necessary if we are to prove that a given program is deterministic.

General (possibly nondeterministic) composite data items are called *webs* and correspond to binary relations on data items. Like fen, webs can be represented by intension. If for some relation on data items, S, we have that the data item w is represented by $[x{:}y \mid S(x, y)]$, then we have that $w[x{:}y] \leftrightarrow S(x, y)$, for all data items x and y. Observe that if F is a function, then $[x \mid F(x)]$ and $[x{:}y \mid y \equiv F(x)]$ are the same fen. It should be emphasized here that the notions of extension and intension apply only to the notation. Thus $[a{:}0, b{:}1]$ and $[x{:}y \mid x{=}a \wedge y{=}0 \vee x{=}b \wedge y{=}1]$ are equal fen.

3. The Semantics of Data Types

As a sample application of the theory we consider the representation of data and the definition of data types. This problem is particularly acute in the area of extensible languages, and a number of attempts have been made to solve it [11, 21, 29, 30]. With the exception of Jorrand [11] and Wirth [29], most work in this area has concentrated on the building of composite data items and data types from certain primitive data items and types. The primitive data types usually include real, integer, Boolean, pointer, and several others. In this section we will present a technique which allows the definition of composite types as well as the usual primitive types, and provides the programmer with a mechanism for defining new primitive types.

3.1. The Construction of Data Types

A data type is a class of data items whose elements (usually) have some structural similarity. The methods used to construct new types from others correspond to the operations permitted in regular expressions. These operations are rowing, structuring, and uniting,[3] symbolized by $*$, $.$, and $\cup$.

The simplest nontrivial data type is that to which only a single data item belongs. Such a type is called a *singleton* data type. If $S_1, S_2, \ldots, S_n$ are a finite number

[1] These notations are left associative.

[2] The relation $\equiv$ denotes equality in the sense of Leibniz, i.e. $a \equiv b \leftrightarrow \forall P[P(a) \leftrightarrow P(b)]$. This is discussed further in Section 3.2.

[3] Terminology is based on that of the ALGOL 68 Report [28].

of singleton data types,[4] then the set of data types constructable is seen to be the set of all regular sets over the alphabet $\sum = \bigcup\{S_i \mid 0 \leq i \leq n\}$.

If T is any data type, then it can be rowed to T^*, the type of all arrays of elements of T. Arrays are modeled by fen whose selectors are integers. If $T_1, T_2, \ldots, T_n$ are data types, then they may be structured to $T_1.T_2.\ldots.T_n$, the type of all composite data items with components of the specified types. Structures are modeled by fen whose selectors are *elementary* fen. A fen is elementary if it has no substructure; i.e. it corresponds to a node from which no edges are directed. If S and T are data types, then they can be united to $S \cup T$, the type to which all elements of either S or T belong. Elements of united types are represented by elements of any of the types forming the union.

3.2. Instances of Data Items

Two concepts of equality are meaningful in the system. Leibnizian, or strong, equality is symbolized by $x \equiv y$ and has the meaning that x and y are identical individuals. This is defined

$$x \equiv y \leftrightarrow \forall P[P(x) \leftrightarrow P(y)].$$

With the above definitions of structured data items, we see that strong equality corresponds to the ALGOL 68 [28] identity relation (:=:). Extensional, or weak, equality is symbolized by $x = y$ and has the meaning that x and y are in the same performing class. Alternately,

$$x = y \leftrightarrow (\forall a, b)(x[a{:}b] \leftrightarrow y[a{:}b]).$$

Thus if x and y are different instances of the same data value, we will have $x = y$ but not $x \equiv y$. Since in most languages different instances of the same value are considered equal, we will probably want the "equal to" relation to correspond to the concept of equality in the language.

4. A Semantic Model of Computation

A formal semantics for a programming language generally has the form of a transformation which takes well formed source language programs into equivalent programs in an abstract language. Often the semantics of the abstract language is specified by an interpreter written in some more primitive language. Sample applications of this technique can be found in [13, 17, 18], and an analysis of the interpretive approach can be found in [24].

The approach to be taken here is similar to that used by Strachey and Scott [26, 27] in that it presents a mathematical (or denotational) semantics for a language. That is to say, meaning is assigned to a procedure by specifying what partial function it denotes. There is no interpreter involved.

[4] This restriction to a finite number of singleton data types is met in all practical languages.

It was mentioned above that there is a close correspondence between the performing class $\{x \mid F\}$ and the lambda expression $\lambda x.F$. In particular the notation $[x \mid F]$ denotes a particular instance of the function $\lambda x.F$. Complex fen can be constructed by substitution in the same way complex lambda expressions can be built by substitution. As an example note that the function

$$Y = \lambda f[\{\lambda x.f(xx)\}(\lambda x.f(xx))]$$

corresponds to

$$y = [f \mid [x \mid f(xx)][x \mid f(xx)]].$$

Statements about *convertability* [5, p. 347] in the lambda-calculus can then, with some exceptions, be translated into statements about strong fen equality. This mechanism enables the computation of recursive functions and the convenient description of applicative (or descriptive, see [14, 15, 16, 27], particularly [16, p. 164–166]) languages.

4.1. Statements and Environment in Imperative Languages

Most languages are not descriptive; i.e. they contain the concepts of an imperative *statement* which can cause the *environment* (state) to change in time. This is formalized by defining statements to be functions on the environment. The environment itself is taken to be a composite data item which contains as fields what we think of as the storage accessible to a series of statements. The symbol C will be used uniformly to indicate the current environment. Following the above definition, if F is any function then $[C \mid FC]$ is a statement.

4.2. The Synthesis of Programs

In [25] Scott has observed that there are three fundamental techniques through which composite programs can be synthesized from simpler ones. We paraphrase these as:

—**product:** the functional product of two statements is a statement.

—**conditional** or **sum:** a fen and two statements can be combined to form a conditional statement.

—**while**-loop: the discussion of **while**-loops is postponed until after recursion has been introduced.

We are dealing with what amounts to *, ., and $\cup$, the operations from which regular expressions are built. It will be shown below that these operations are easily defined as fen.

4.3. The Product of Statements

It will be convenient at this point to introduce a notation for non-unary fen. We define $[x_1x_2 \ldots x_n \mid F]$ to be $[x_1 \mid [x_2 \mid \ldots [x_n \mid F] \ldots]]$. Thus $[xy \mid Fxy]$ means $[x \mid [y \mid Fxy]]$ and corresponds to the lambda expression $\lambda xy.Fxy$. Similarly we write $F[x_1, x_2, \ldots, x_n]$ for $F[x_1][x_2] \ldots [x_n]$.

The right product (composition) of f and g will be denoted by $f;g$. This operator is formally defined by

$$; = [rs \mid [C \mid s[rC]]] = [rsC \mid s[rC]].$$

This operator is declared to be left associative to allow statements separated by semicolons to mean what we intuitively expect; e.g. $(f;g;h)C = h[g[fC]]$.

4.4. The Sum of Statements

The conditional is represented in most languages by a construct resembling **if** B **then** T **else** F, where B is a fen and T and F are statements. The intuitive meaning is that if $BC =$ **true**, then evaluate TC, whereas if $BC =$ **false**, then FC will be evaluated. The formal definition of a conditional is as follows.

Definition. If B is a fen and T and F are statements, then **if** B **then** T **else** F stands in the place of $[C \mid [\mathbf{true}{:}T, \mathbf{false}{:}F][BC]C]$.

The following theorem states that this definition works as expected.

THEOREM. *If* $BC = \mathbf{true}$ *then* (**if** B **then** T **else** F)$C = TC$ *and if* $BC = \mathbf{false}$ *then* (**if** B **then** T **else** F)$C = FC$.

PROOF. We prove the theorem for $BC = \mathbf{true}$ only, as the other case is exactly analogous. Suppose $BC = \mathbf{true}$, then

$$\begin{aligned}
&(\mathbf{if}\ B\ \mathbf{then}\ T\ \mathbf{else}\ F)C \\
&\quad = [C \mid [\mathbf{true}{:}T, \mathbf{false}{:}F][BC]C]C \\
&\quad = [\mathbf{true}{:}T, \mathbf{false}{:}F][BC]C \\
&\quad = TC \ \square
\end{aligned}$$

It should be noted that $BC = \mathbf{true}$ (or **false**) requires that the computation of B on C will terminate. If this is not the case, then BC will have no value.

4.5. Assignment

If for the moment we assume that several operators are defined in the obvious way, then we might write a small program as follows:

$$\begin{aligned}
&y \leftarrow [Hd{:}0, Tl{:}[Hd{:}1, Tl{:}[Hd{:}2, Tl{:}Nil]]]; \\
&x \leftarrow Cy[Tl][Hd]; \\
&z \leftarrow Cy[Tl][Tl][Hd]; \\
&w \leftarrow Cx + Cy.
\end{aligned}$$

If R is the result of this program applied to an elementary item, then $Rw = 3$. In this context the prefix C is seen to be analogous to a dereferencing operator on the environment.

The assignment, or updating operator, $\leftarrow$, is defined as follows:

$$\begin{aligned}
&x \leftarrow y[C] = Update[C,x,y] \\
&\text{where } Update = [Cxy \mid [a{:}b \mid c[a{:}b] \ \wedge \ a \not\equiv x \ \vee \ a \equiv x \ \wedge \ b \equiv y]].
\end{aligned}$$

Defining $Update$ in this way causes the allocation of x in C if it is not already a selector of C.

The following elementary theorem guarantees that the assignment operator has the properties we expect it to have. This theorem corresponds to Kaplan's axioms A1 and A2 in [12].

THEOREM. *For any fen* F, C *and selectors* x, y *where* $x \not\equiv y$, *we have* $Update[C,x,F]x = F$ *and* $Update[C,x,F]y = Cy$.

PROOF. Let the hypothesis be satisfied.

$$\begin{aligned}
&Update[C,x,F]x \\
&\quad = [Cxy \mid [a{:}b \mid c[a{:}b] \ \wedge \ a \not\equiv x \ \vee \ a \equiv x \ \wedge b \equiv y]] \\
&\quad CxFx \\
&\quad = [a{:}b \mid c[a{:}b] \ \wedge \ a \not\equiv x \ \vee \ a \equiv x \ \wedge \ b \equiv F]x \\
&\quad = F.
\end{aligned}$$

Similarly the other result holds. $\square$

One consequence of this model is that only statements can have side effects in the environment. A subexpression of a statement can affect the local environment only indirectly, i.e. by passing a value to the statement of which it is a part. A result of this is that the assignment operator has no value.

4.6. The Star of Statements

To model computation in any reasonable way, it is necessary to be able to represent iteration or recursion. The approach to recursion taken here is based on the technique used in real implementations; i.e. the function is part of the environment. We indicate the general technique with an example, the factorial function. Again, any undefined fen should be assumed to have their obvious definitions.

$$\begin{aligned}
&x \leftarrow 0;\ y \leftarrow 10; \\
&\text{fact} \leftarrow [Cx \mid \mathbf{if}\ x = 0\ \mathbf{then}\ 1\ \mathbf{else}\ x{*}C\ \text{fact}\ [C, x]]; \\
&x \leftarrow C\ \text{fact}\ [C, Cy]
\end{aligned}$$

At termination x will select 10! from the resulting environment. Observe that the definition invokes fact in the environment of the caller.

Recursion is often introduced into formal systems with Curry and Feys' *paradoxical combinator*, Y, which is defined by

$$Y = [f \mid [x \mid f(xx)][x \mid f(xx)]].$$

This approach is taken by Strachey [27], Henderson [8], Landin [13], and Böhm [2].[5] The definition of Y is such that $Yf \rightarrow fYf \rightarrow ffYf \rightarrow \cdots$.

The Y combinator is seen to be a special case of storing the function in the environment. With the Y combinator the local environment contains only the function $[x \mid f(xx)]$. The fen approach is to be preferred since it is more in accord with actual practice. Also, as is shown later, this is itself a special case of the more general problem of environment.

4.7. While-loops

We briefly discuss the semantics of the **while**-loop. The intuitive meaning associated with **while** B **do** S is that the statement S is to be applied to the environment as long as the fen B evaluates to **true.**

Let W be a selector distinct from all those used in a given program. We assume each environment contains

[5] Böhm calls Y by θ, following Rosenbloom.

what amounts to the result of the assignment:[6]

$W \leftarrow [CBS \mid (\textbf{if } B \textbf{ then}(S;\textbf{while } B \textbf{ do } S) \textbf{ else } KC)C]$.

The **while**-loop is defined by stating that the statement **while** B **do** S stands in the place of the statement $[C \mid CW[C, B, S]C]$. Thus the **while**-loop operates by recursively calling itself as long as B on the current environment is **true**. Following is a proof that this definition works as intuitively expected.

Theorem. *If B, C are fen and S is a statement and $B(S^nC) =$ **false** and for all $k < n$ (with $k \geq 0$) we have $B(S^kC) =$ **true**, then (**while** B **do** S)$C = S^nC$.*

Proof. Suppose the hypothesis is satisfied. The proof will be by induction on n. For the case $n = 0$ we are assuming $BC =$ **false**. By the definition we have

$$\begin{aligned}(\textbf{while } B \textbf{ do } S)C &= CW[C, B, S]C\\ &= (\textbf{if } B \textbf{ then } (S;\ \textbf{while } B \textbf{ do } S)\textbf{else } KC)C\\ &= KCC = C.\end{aligned}$$

Now suppose $n > 0$. Since $B(S^nC) =$ **false**, we know $B(S^{n-1}(SC)) =$ **false**. Since for all $k < n$ $B(S^kC) =$ **true**, we know for all $j < n - 1$, $B(S^j(SC)) =$ **true**. Since $B(S^nC) =$ **false**, we know SC terminates and is thus a fen. By the induction hypothesis

$$\begin{aligned}(\textbf{while } B \textbf{ do } S)SC &= S^{n-1}(SC) = S^nC.\\ (\textbf{while } B \textbf{ do } S)C &= CW[C, B, S]C\\ &= (\textbf{if } B \textbf{ then } (S;\ \textbf{while } B \textbf{ do } S) \textbf{ else } KC)C\\ &= (\textbf{while } B \textbf{ do } S)SC\\ &= S^nC \ \square\end{aligned}$$

The following two corollaries follow trivially from the above theorem. Together they constitute Hoare's axiom D3 [10].

Corollary. *If for some finite n, $B(S^nC) =$ **false**, then (**while** B **do** S)$CB =$ **false**; i.e. the condition is false after the loop.*

Corollary. *If for some finite n, $B(S^nC) =$ **false**, and for all fen, C, and predicates, P, we have that if PC then $P(SC)$, then we have that if PC then $P(($**while** B **do** $S)C)$.*

4.8. Environment

Most programming languages include some concept of "scope of names" and automatic allocation and deallocation of storage. Change of scope and storage allocation/deallocation is usually associated with entry to and exit from *blocks*, composite statements delimited by **begin** and **end** or similar symbols. The semantics associated with **begin** typically include pushing down the current environment and creating a new environment with some preallocated variables. The semantics associated with **end** involves deleting the current environment and restoring the environment saved by the last **begin**.

Some mechanism is provided to allow statements to have access to the *global*-environment, the collection of environments saved by as yet unmatched **begin**s. The following discussion presents two ways in which the global environment can be maintained.

Previous environments can be maintained as a linked list with the local environment as the head of the list. Let $K = [xC \mid x]$; i.e. K is a statement which makes its argument the new local environment. If we then define

begin $= K[[global{:}C]]$

the statement **begin** will create a new local environment whose only selector is *global*, which contains the old environment.[7] If we define

end $= K[Cglobal]$

then the **end** statement will have the effect of restoring the old environment. References to a variable x in the local environment are accomplished, as before, by Cx. References to x in the global environment are accomplished with $Cglobal\,x$, $Cglobal\ global\,x$, etc., depending on how deeply the environment of x has been depressed. Updates of the global environment are more complicated, as the following example indicates.

```
begin;
  $x \leftarrow 1$;
  $z \leftarrow 1$;
  begin;
    $y \leftarrow 2$;
    $global \leftarrow (z \leftarrow Cy + Cglobal\,x)[Cglobal]$;
  end;
end
```

If S is any product of statements then the construction $G \leftarrow (S)[CG]$ is called an *environment switch* and may be abbreviated $G{:}S$. In the above example we could have written

$global{:}\ z \leftarrow Cy + Cglobal\,x.$

As suggested above, chaining is not the only mechanism through which access to global environments may be obtained. Another possible technique is to maintain in the local environment a vector[8] of previous environments. Accesses to the global environment for x take the form $CE[1]x$, $CE[2]x$, etc. Both of these methods correspond closely to real implementation techniques.

5. Conclusions

An abstract entity (called a web) has been introduced which corresponds to a particular instance of a relation. Some of these entities (those called fen) correspond to particular instances of functions. The resulting mechanism is so general that one can summon up the power of the algebras of classes and relations, if this is necessary. Furthermore, the system is close enough to the basic substratum of computer science to enable concise definitions of important semantic concepts. Common infor-

[6] The statement KC, as will be shown later, is a do-nothing or empty statement.

[7] This technique is analogous to that used by Landin in [13] in which the selector *global* was called D (*Dump*).

[8] A vector is a fen whose domain is a contiguous subset of the integers.

mation structures such as arrays, records, random files, relational data bases, and functions are seen to be special cases of webs and fen.

As sample applications of the theory, a possible semantics for data types and computations was presented. Clearly the models presented are not the only ones possible, or even desirable. The semantics of some languages might require a data item to include an indication of its type. Similarly, certain languages will allow side effects. These will have to be allowed for by making each value a two-element structure, one of whose fields is the environment (see [27]).

It is not obvious that a lambda-calculus format is the best representation of a program. In [7] and [19] it has been suggested that a format similar to that of a Petri net would be superior for optimization and machine independent compiler output. The theory of webs and fen lends itself nicely to the description of this variety of computation, and work is currently being done on the complete formal specification of a language designed on this basis. When it is clear which semantic primitives are necessary, a microprogrammed implementation is intended. It is hoped that the formal system will provide a convenient medium in which to prove the correctness both of the implementation and of programs based on the implementation.

Appendix

The Formalism

The following presentation of the formalism governing data items (webs and fen) assumes some familiarity with formal systems as defined by Curry and Feys in [6, ch. 2].

Morphology

1. Primitive Ideas. The obs of the system can be classified as follows.

The connectives $\neg$, $\rightarrow$, ϵ, and the punctuation symbols (and). Using these symbols and well known definitions the usual formulas of the predicate calculus can be constructed.

The system includes a denumerable infinity of individual variables serving as place holders for names of members of the domain of interpretation. On the intended interpretation these variables will stand for arbitrary "things."

The system includes one ternary predicate constant, $w[x{:}y]$. On the intended interpretation $w[x{:}y]$ means the selector/field pair $x{:}y$ is a member of the ob w.

2. Formation Rules. Individual variables are terms.

If w, x, and y are terms, then $w[x{:}y]$ is a formula.

If R and S are formulas, then $\neg\ R$ and $R \rightarrow S$ are formulas.

If x is an individual variable and S is a formula, then $(\epsilon x)S$ is a term. The intended meaning of $(\epsilon x)S$ is "any x such that S."

The application of the above rules will be further constrained by the requirements of simple type theory.

Using the above formation rules and obvious definitions all statements concerning the obs of the system can be expressed.

Transformation Rules

1. Axioms. The axioms of the system include the four usual axioms of the propositional calculus, the axiom governing the ϵ operator [9], and the axioms commonly known as extensionality and Zusammenfassung. These axioms provide for the usual theorems of logic and the algebra of classes.

There are two axioms that govern the individuals of the system.

The term *web* is introduced to denote the formal concept corresponding to the informal concept of "data item."

Definition. A web is any "thing" (individual) which satisfies:
—at least one individual is not in its domain, and
—at least one individual is not in its range.

This distinction between a web and a "thing" is exactly analogous to the distinction between a set and a II.Ding in von Neumann's set theory [22, 23]. The exclusion from web-hood of individuals that have domains or ranges containing the universe prevents the definition of contradictory webs.

Definition. We say that an individual, w, *performs* a relation, R, when for all x and y, $w[x{:}y]$ if and only if x and y are webs and $R(x, y)$. The class of all webs (n.b. *not* individuals) that perform a given relation is called the *performing class* of that relation.

Since webs correspond to data items in the intended domain of interpretation, and since it is intuitively felt that there should be an unlimited number of "copies" of any datum, we postulate:

AXIOM 1. *WEAX—the Web Existence Axiom. If R is any relation on individuals, then the performing class of R is infinite.*

This axiom will guarantee that a sufficient number of webs are available to derive the theorems we desire to be true. Note that it is not required that the performing class of a relation be denumerably infinite.

The *field* of an individual is defined to be the union of its domain and its range. The following theorem is easy to prove using the definition of performing.

THEOREM. *The field of an individual contains only webs if and only if there is some relation (on webs) that that individual performs.*

This theorem suggests that we should postulate:

AXIOM 2. *WFAX—the Web Field Axiom. The field of an individual contains only webs.*

2. Rules of Inference. The rules of inference are Modus Ponens and generalization. Thus conventional techniques may be used in derivations and proofs about webs.

Received April 1973

References

1. Bekić, H., and Walk, K. Formalization of storage properties. In *Symposium on Semantics of Algorithmic Languages*. Erwin Engeler (Ed.), Springer Lecture Note Series, Springer-Verlag, Heidelberg, 1971, p. 39.
2. Böhm, C. The CUCH as a formal and descriptive language. In *Formal Language Description Languages for Computer Programming* T.B. Steel (Ed.), North-Holland, Amsterdam, 1966, pp. 198–220.
3. Cadiou, J.M., and Manna, Z. Recursive definitions of partial functions and their computations. SIGPLAN Notices 7 (Jan. 1972), 58–65.
4. Church, A. *The Calculi of Lambda-Conversion*. Princeton U. Press., Princeton, N.J., 1951.
5. Church, A. An unsolvable problem of elementary number theory. *Am. J. .Math. 58* (1936), 345–363.
6. Curry, H. B., and Feys, R. *Combinatory Logic, Vol 1*. North-Holland, Amsterdam, 1958.
7. Dennis, J. B. Notes on the design of a common base language. Mimeographed notes prepared for Tutorial Symp. on Semantic Models of Computation, Dec. 1971.
8. Henderson, P. Derived semantics for some programming language constructs. *Comm. ACM 15*, 11 (Nov. 1972), 967–973.
9. Hilbert, D., and Bernays, P. *Grundlagen der Mathematik*, Berlin, 1934 and 1939, Sec. 1.
10. Hoare, C.A.R. An axiomatic basis for computer programming. *Comm. ACM 12*, 10 (Oct. 1969), 576–583.
11. Jorrand, P. Data types and extensible languages. SIGPLAN Notices 6 (Dec. 1971), 75–83.
12. Kaplan, D.M. Some completeness results in the mathematical theory of computation. *J. ACM 15*, 1 (Jan. 1968), 124–134.
13. Landin, P.J. The mechanical evaluation of expressions. *Computer J. 6* (Jan. 1964), 308–320.
14. Landin, P.J. A formal description of ALGOL 60. In *Formal Language Description Languages for Computer Programming*. T. B. Steel (Ed.), North-Holland, Amsterdam, 1966, pp. 266–294.
15. Landin, P.J. A correspondence between ALGOL 60 and Church's lambda notation. *Comm. ACM 8*, 2 (Feb. 1965), 89–101, and (Mar. 1965), 158–165.
16. Landin, P. J. The next 700 programming languages. *Comm. ACM 9*, 3 (Mar. 1966), 157–166.
17. Lee, J.A.N. *Computer Semantics, Studies of Algorithms, Processors and Languages*. Van Nostrand Reinhold, Princeton, N.J., 1972.
18. Lucas, P., and Walk, K. On the formal description of PL/I. *Annual Reviews of Automatic Programming 6*, 3 (1969).
19. MacLennan, B. Semantic specification and machine independent compilation. Proc. ACM 10th Ann. Southeast Reg. Conf., June 1971 (mimeographed).
20. Manna, Z., Ness, S., and Vuillemin, J. Inductive methods for proving properties of programs. SIGPLAN Notices 7 (Jan. 1972), 27–50.
21. Morris, J.B. Jr., and Wells, M.B. Generalized data structures in Madcap VI. SIGPLAN Notices 6 (Feb. 1971), 321–336.
22. von Neumann, J. Die Axiomatisierung der Mengenlehre. *Math. Z. 27* (1928), 669–752.
23. von Neumann, J. Eine Axiomatisierung der Mengenlehre. *J. reine angew. Math. 154* (1925), 219–240.
24. Reynolds, J.C. Definitional interpreters for higher-order programming languages. Proc. ACM 25th Nat. Conf. 1972, pp. 714–737.
25. Scott, D. The lattice of flow diagrams. In *Symposium on Semantics of Algorithmic Languages*. Erwin Engeler (Ed.), Springer Lecture Note Series, Springer-Verlag, Heidelberg, 1971, pp. 311–366.
26. Scott, D. and Strachey, C. *Toward a Mathematical Semantics for Computer Languages*. Oxford U. Computing Lab., Programming Research Group Tech. Mono. PRG-6.
27. Strachey, C. Towards a formal semantics. In *Formal Language Description Languages for Computer Programming*. T.B. Steel (Ed.), North-Holland, Amsterdam, 1966, pp. 198–220.
28. van Wijngaarden, et al. Report on the algorithmic language ALGOL 68. *Numerische Mathematik 14* (1969), 79–218. Offprint available from Dept. AL68-D, ACM Headquarters, New York.
29. Wirth, N. The programming language Pascal. *Acta Informatica 1*, 1 (1971), 35–63.
30. Wulf, W.A., et al. *Bliss Reference Manual*. Dept. of Computer Sci. document, Carnegie-Mellon U., Pittsburgh, Pa., 1970.

Petri Nets and Speed Independent Design

David Misunas
Massachusetts Institute of Technology

Petri nets are investigated as one method of modeling speed independent asynchronous circuits. A study of circuit realizations of Petri nets leads to a demonstration of their usefulness in modeling speed independent operation. This usefulness is emphasized by the design of a speed independent processor from modules developed in the investigation of Petri net implementation.

Key Words and Phrases: speed independent asynchronous, Petri net
CR Categories: 6.1, 6.33

1. Introduction

The utilization of asynchronous techniques in logic design has introduced many new problems in the digital field. One of the difficulties inherent in the design of asynchronous circuits is the possible presence of arbitrary delays both in elements and connections of the circuit. A circuit in which such delays have no effect upon circuit operation, other than possibly causing the circuit to run slower, is known as a speed independent circuit.

Early studies of speed independent circuits [10, 19, 22] were concerned with methods of modeling such circuits and describing their operation in some precise mathematical form. These studies merely exposed the problems inherent in developing such a method. Any means in order to be useful in depicting speed independent operation must: (1) contain a direct correlation be-

Author's address: MIT project MAC Room 530, 545 Technology Square, Cambridge, MA. 02139.