9 The algebraic specification of a target machine: Ten15

J. M. Foster, Royal Signals and Radar Establishment

9.1 Introduction

Ten15 is a formally defined abstract machine and a target for high-level language compilers. It provides a basis for formal methods over the whole area of programming, from operating systems to users' programs, from assembler-like constructions to high-level languages, and from simple text construction of programs to elaborate transformation systems. It also provides a practical basis for creating programs and for program manipulation. So it is something on which to found a programming environment. Meeting these requirements together is not easy, but there have been enough advances in recent years for this to be a timely moment to make the attempt. Experience and practical use of Ten15 have demonstrated the possibilities. This chapter considers the requirements and describes in outline how Ten15 has met them.

It is not to be expected that one class of formal methods will satisfy all needs. At present there are groups of methods with various foundations, particularly in predicate calculus and algebra, and within these groups there are competing techniques. All of these techniques are needed in our armoury. Ten15 is intended as a groundwork for as many of them as possible, so that tools using different methods have at least some common factors. This will be of advantage to avoid the duplication of work and to enable tools to cooperate with each other. Ten15 does not make a break with current practice and start afresh; green field solutions to old problems face almost insurmountable difficulty in becoming accepted, and it is unwise to discard systems which have shown themselves to have at least some of the needed characteristics. It is aimed at current methods of producing practical programs, current operating systems and current problems, while avoiding constraining the introduction of new methods. Practical usefulness is paramount. Ten15 is something out of which people build a variety of systems, not a single Procrustean system, to which users have to fit their problems.

C. T. Sennett (ed.), *High-Integrity Software*

In outline, the requirements are these. Ten15 needs to be expressive enough to account for all the necessary kinds of programming and data. It needs to be well matched to the way people want to think about programs, because at times it will have to serve as an interface between tools and users. It needs to cover a wide spectrum, from high- to low-level constructs, if it is to serve as a basis for transformations. It must be possible to translate it into efficient code for conventional machines, since it is to be used. It needs to be in the intersection of predicate calculus and algebra, since it is to serve as a basis for tools based on each of them.

9.1.1 Expressive range

Programming languages started with assemblers, autocodes and Fortran, and have proliferated wildly. Only a few established themselves generally; each of these has serious shortcomings, and there is no universal language available, nor is there likely to be. The very heat and passion of the debate between languages and its eternal fruitlessness is evidence of the need to retain a choice. The best language depends on the problem. There are problems for which Ada is better than Lisp, and problems for which Lisp is better than Ada. Problems which cut across such categories are in difficulty. A programming environment needs to have a number of languages available, and to have the languages capable of working together, in the sense that systems can be made out of components written in different languages and such systems can be analysed.

Operating systems developed on different lines from application programs. Early operating systems were introduced in order to make the best use of the computer, and to provide standard ways of driving peripherals and allocating resources. They evolved by adding features to make this more convenient for users, particularly by introducing job control languages. These languages were developed *ad hoc*, and were interpreted, not compiled. It has often been remarked that they were programming languages and should be considered as such, and that considered as languages they were crude. But progress along this line has been disappointing.

The programming of the operating system itself has usually been done in a mixture of assembly code and high-level language, the high-level language wherever possible and the machine code for the parts which could not be expressed in the language or which would be too slow if they were. Operating systems and the corresponding programs in operational computers are among the most important programs to be analysed, since on them the correctness of all other programs depends. No matter how well a program is validated in itself, if the operating system wrongly allocates store to it or to other programs then that validation is destroyed, and the errors that can be introduced may be very difficult to diagnose.

Ten15 is intended not only to provide a basis for conventional users'

programming but also for the programming of operating systems and for the equivalent of job control language, so that analysis and manipulation can be carried out on all these classes of program.

The need to support operating systems as well as user programming forces Ten15 to include types and operations which are adequate to express the extra features. For example, types to handle database and network values are essential. The kinds of value which conventional systems handle on backing stores and in databases are curiously rudimentary. It is common that the only method of communicating between programs is through files, the structures of which are closely related to the physical properties of disc drives. In some cases sequences of lines of characters are the only data structure supported and dictionaries provide the only way of accessing them. Even in more sophisticated databases the data structures are closely tied to the specific view of data which that particular system uses, and are usually provided in order to implement individual and elaborate mechanisms. Ten15 provides types and operations which can be composed to make such systems, while retaining the advantages of type checking. The data structures on backing store need not differ very greatly from those used in main store. Certainly the relative speeds of operations are different in the two cases, but the needs are much the same. In terms of an appropriate set of types for backing store all the conventional types of data management system may be programmed and more besides.

Similar remarks apply to networks. The growing use of remote procedure calls [Xerox1981, Foster/Currie1986, Spector1982] suggests that for loosely coupled networks many of the kinds of data familiar from main store can be used. A number of important problems arise here, outside the scope of this chapter [Currie/Foster1987].

Operating systems have to handle users' programs as data, since they have to be able both to allow for their creation and to supervise their running. Procedures and processes are the basic ideas here, and this implies that true procedure values are needed. Operating systems also have to provide debugging facilities; this is an interesting area to treat formally.

A characteristic of operating systems and of the control programs for operational computers is that they contain deliberately non-terminating programs. A command line interpreter is an example; it interprets every line as it is presented and goes on doing this indefinitely. The formalism has to deal with this situation.

Concurrency is another topic which must be included. Unfortunately, pseudo-parallel processing on one computer, loosely coupled networks of computers and tightly coupled networks each seem to require different primitives [Hoare1978, Milner1980]. At present Ten15 does not include methods of treating tightly coupled networks and research in this area is continuing. The problem is not to provide the primitives for tightly coupled networks, but to unify the treatment of all types of concurrency.

9.1.2 Structured for people to use

In spite of the deficiencies of current languages, it would be generally agreed that there has been progress in their design. One important area of improvement has been in the use of types. Types were originally introduced because compilers needed to know the size of objects and the meaning of operations on them in order to produce the appropriate code. But other advantages soon became clear. The types introduced in Algol60 provided checks that found many errors in programs at compile time. The code that was produced did not have to check for these errors at run time and so could be more compact and faster. There were only a fixed number of types in Algol60—ignoring the dimensionality of arrays—but later languages, for example Pascal, introduced the generation of indefinitely many types. Neither Pascal nor Algol had a system of types which made it easy to be sure at compile time that there were no type errors, since parameter types were not fully specified. The importance of strong type checking was realized later.

Strong type checking guarantees that, on any path through a program, operations expecting arguments of a particular type are applied only to values of that type, which provides a powerful check at compile time against certain errors. But this fact can be considered in a different way. Type checking guarantees that only operations expecting type X values can be applied to a type X value, but it also ensures that a value of type X must have been generated by one of the operations which produces type X values and by no other operation. This means that it is relevant to integrity and to information hiding. For example, suppose that a reference is passed from one program regime, P, to another, Q. Consider a type system in which there are values of type *reference*, which can be written and read, and values of type *read-only reference*, which can only be read. Let there also be an operation which converts references into read-only references, without otherwise modifying them, but no operation to perform the reverse transformation. Then if P applies this operation to one of its references and passes the result to Q, it can be sure that Q can at any time read the contents of the read-only reference, but cannot ever alter it, because of the absence of operations to do so. This particular access control is similar to that which can easily be provided on capability machines or in file stores, where the operations are interpreted. But the type mechanism is potentially a much more refined method of access control, and it operates at compile time, so avoiding the interpretive overhead.

The concept of types is so important and useful that it must be considered as a requirement of Ten15. This presents a difficulty, since the type systems of the various languages in use have been developed in isolation from each other, and are in any case each inadequate for the total job. Furthermore, work on possible systems of types has advanced considerably beyond the

systems used in well known languages [Reynolds1985, MacQueen/Plotkin/Sethi1986]. So Ten15 must have a global system of types into which the types in existing languages can be mapped, which is capable of describing all the necessary constructions efficiently, and which may be extended as new needs arise.

Another tendency in the development of programming languages has been the invention of constructions which allow programmers to separate decisions. Perhaps the first example of this was the use of labels in assembly code, so that one did not have to give the absolute address of the instruction which was to be the destination of a jump, but could delay filling in that address until the program was assembled. Such constructions usually permit decisions to be delayed, but it would be better to speak of controlling the time of taking decisions, since it is often separating the decisions, rather than delaying them, which is important. The skill of programming may be regarded as the choice of the right moment to take decisions. Examples of this abound. The identification of variables, the introduction of procedures, arrays of dynamically chosen size, heap storage and separate compilation are all instances of this development. It is of the greatest importance that Ten15 should facilitate such separation.

Among the most important features of operating systems is the use of interactive working, both on multi-user systems and on single-user workstations and personal computers. Here is another case of the delaying of decisions, in this case the decision of what to do. The characteristic of interactive working is that at the start of a session the user does not know exactly what to do, which procedures will be applied to which values, and in what order. Constraints which prevent programs from working together, or which funnel interactions through a narrow data channel, such as converting the output of one program into a text file and then recreating a similar data structure from it in the next program, are quite against the style of interactive work. Data must pass directly between programs, the combination of which was not anticipated, and this without compromising the integrity of the programs or the system. Clearly, common store is needed, so the question of integrity is raised. Strong type checking is exactly the mechanism needed to permit and control such access.

9.1.3 Wide spectrum

In the past, programs were created by writing a text version and then compiling this into machine code. Separate compilation complicated this picture only slightly; the running program was still produced by the compiler from texts, and it was only the compiler that produced programs. Indeed one acted as if the text was the program. More recently, it has become common to produce programs by other means. For example, in the Flex system [Foster/Currie/Edwards1982], many programs exist to which it is not

natural to attribute a text version, since they were not created by compiling text, but by applying other program-creating programs to non-text data.

An example of this tendency is the creation of programs by program transformation, which has been a research topic for many years. One might want to write a program in a clear way which can be seen to be correct, and transform it, by methods which are known to produce programs of equivalent effect, into a version which will run with adequate efficiency. If this is to be done, all the various versions of the program must be in a formalism which can be manipulated by the transforming program. This needs to move continuously from high-level constructs down to constructs which correspond sufficiently closely with conventional machines to be fully efficient. Ten15 must have a wide spectrum of constructions.

9.1.4 Efficiently translatable

Program analysis needs to be tied as directly as possible to the program which actually runs. If an Ada program is translated into Ten15 and analysed and then taken away to another system to be translated with an independent Ada compiler, some confidence in the original analysis has been lost. The Ada program or the operating system of the distant computer may not exactly correspond to the way Ada was translated into Ten15. If instead Ada is translated into Ten15 and then that same Ten15 into machine code, this particular discrepancy does not exist. Neither Ada nor any other language is sufficiently well specified for anyone to be confident that there are no such differing interpretations. The same point arises even more strongly for mixed language working or the production of programs by transformation. In that case Ten15 may be the only available expression of the program from which code is to be produced. So the Ten15 constructions must be directly translated into machine code. If this is to produce real operating systems the resulting code on conventional computers must be efficient. Certainly it is permissible to pay a small price in speed and space for the extra facilities and confidence provided by Ten15, but only a small one. The amount is arguable; perhaps 25% is tolerable.

9.1.5 Formal aspects

One result of the textual bias of compiling systems has been too great an emphasis on syntax. The syntax of a language is only there to provide a representation for programs for an underlying machine. The bones of a language are the operations of that underlying machine; the syntax is cosmetic. Too often gross complexity is introduced in the syntax and explained in terms of it, but only an elementary machine lies behind the facade. Though the syntaxes of languages vary greatly, in many cases the

algorithmic constructions behind them are very similar. Procedures, conditional statements, loops, subscripted arrays and variables are common to most languages. If one is manipulating or proving properties of programs it is these essential constructions with which one is concerned, rather than the details of the text representing them. Of course, the manipulation has to be in terms of some representation, but the linear text string of letters is unlikely to be the appropriate one. A representation which reveals the structure is appropriate. For many years the term "abstract syntax" has been used for something like this concept, but it has not been well defined and it tends to be associated with specific languages, in the sense that one might speak of the abstract syntax of Pascal or the abstract syntax of Ada.

A curious byproduct of the textual bias has been the use of macros. Macros were introduced early into programming languages as a method of doing textual substitution before compiling. However, they did not have to respect the syntactic structure of the program so long as the final output of the macro stage was syntactically correct. This led to some very rebarbative and impenetrable programming techniques. It is often useful to carry out systematic substitutions in programs before they are compiled, but these should form a part of the process of generating programs by transformation and should respect the structure of programs.

Whether tools are based on predicate calculus or on algebra, they need some way of describing programs which corresponds to the essential structure of the program. Consider, as an example, conditional expressions. Any particular conditional expression has a number of sub-expressions, and the meaning of the whole is related to the meanings of the parts in a particular way which is characteristic of this construction. In predicate calculus this relation will be described by means of predicates and in algebra by algebraic means.

For the definition of an abstract machine, the atomic program constructions and the ways of combining them to produce more elaborate pieces of program have to be specified. Giving a definition of these structures is the same as giving a semantics of the machine.

The commonest methods of defining the semantics of machines have been to give definitions by means of denotational semantics, weakest preconditions or operational semantics [Scott1970, Dijkstra1976]. These methods all study the meaning of the evaluation of a program. Each becomes more complex when applied to non-terminating programs and procedure values. Instead of using one of these methods, the definition of Ten15 is given by developing the whole of a program, as an infinite object, and defining which programs are equal. This method uses laws as in Hoare et al [Hoare1987] but in this case to give a formal and complete definition. The intended significance of equality is that programs are equal if and only if they behave identically in all possible circumstances. This, however, is only the intuitive meaning; the definition of equality is given purely formally. In order to do this some constructions have to be introduced into Ten15

especially for the purpose. The determination of the equality of Ten15 programs is not a computable problem, nor is this desirable, for if it were computable Ten15 would not be sufficiently expressive. But it is possible to isolate parts of Ten15 where equality is computable, and indeed this is part of the process of showing that Ten15 is adequately defined.

Later sections will show how the Ten15 algebra is defined inductively. The basic algebra is rather like a word algebra with extra constructions. A congruence is then given on this algebra and Ten15 is identified with the quotient of this algebra by the congruence relation. Many properties may be evaluated by means of homomorphisms on the algebra, and by other algebraic means. This definition lies easily within a predicate calculus formulation and so allows predicate calculus based tools to be used. Whether the form of the definitions will prove intuitively easy to use remains to be seen.

9.1.6 Present state

Ten15 is an abstract machine, formally defined as an algebra. It is the target of a number of compilers which are in various stages, including compilers for Ada, Pascal and Algol68, and more are planned. It can also be produced by transformations and other non-compiler tools. It can be translated into machine code for conventional computers, at present Vax and the Flex computer. Programs for Ten15 can be the object of manipulation, by both algebraic and predicate calculus methods. It supports the range of programming discussed above.

Some of the manipulations of Ten15 that are needed have been discussed, in particular compilation into machine code and the transformation and combination of programs to yield new ones. Many other kinds of tools will be needed and, though it would be against the spirit of this chapter to think that an exhaustive list could be given, it would be useful to skim through some of the possibilities. Possible tools, in no particular order, are: proof of correctness, proof of properties, examination for deadlock, symbolic evaluation, showing that a program lies in a particular class such as time-bounded programs, showing that particular stylistic programming rules have been met, producing a concordance, producing other program annotations to help in understanding it, showing that overflow cannot occur, showing that there can be no error in indexing arrays, finding the places in a program that would be affected by a proposed change, producing an abstract program that can be specialized to particular uses in a systematic way, and making systematic changes to a program that can only be done if its structure is known.

The rest of the chapter discusses the type system, the Ten15 machine and its formal definition.

9.2 Types and operation

Strong typing has been used in many programming languages, but not in a completely uniform way. Different languages have had different ideas of what types mean.

Ten15 regards types as sets of values. Types are used for a number of different purposes: to give control over access, to guard against programming errors of certain kinds, and to allow the translator to deduce constraints on the program that allow it to produce better code. The types of conventional programming languages must be mapped into the Ten15 types, in the sense that values belonging to the Ten15 types serve to represent values operated on by the programming languages. The needs are not antagonistic, but the conclusions to which they lead have to be merged together into a coherent system.

The types are related to the operations in an obvious way. If an operation is said to take a parameter of type X it must be applicable to all values of that type. If an operation is said to deliver a type, then all values that it can deliver must belong to that type. These rules do not determine which sets can be chosen as types. The choice is a pragmatic one.

Consider some simple examples. Suppose that there is a conditional expression, yielding an integer answer

if *a<10* **then** *2*a* **else** *a/2* **fi**

The type of the result depends on the original value of *a*. Suppose that *min-evenint* is twice the integer part of *minint/2*. If the original value of *a* lay between *minint* and 10, then the values less than half *minint* will cause an overflow and so will not appear here. The remaining ones will produce all the even values between *min-evenint* and 18. If the original value of *a* lay between 10 and *maxint*, the result will be all the integers between 5 and the integer part of *maxint/2*. Thus the result values considered as a function of the value of *a* are given by the following expressions:

$$a \in \mathit{minint}, \mathit{minint}+1, \ldots 9 \rightarrow \mathit{min\text{-}evenint}, \mathit{minevenint}+2, \ldots 18$$

$$a \in 10, 11, \ldots \mathit{maxint} \rightarrow 5, 6, \ldots \mathit{entier(maxint/2)}$$

What can be said about the type of the conditional expression? The type could be a function from the state of the machine to possible sets of values, which is approximately what is written above. The disadvantage of this approach is that it discards no information: the type theory is no different from the semantics of the machine. Indeed, if general loop constructions are introduced the type is not in general computable. But the types must be deduced at compile time from the program, and the above sense of type will not allow this.

The input state information could be discarded, giving a type like

min-evenint, min-evenint+2, ... 5, 6, ... maxint/2

but even this formulation is too difficult to handle. It is quite a complex set description in itself, and for arbitrary expressions the complexity increases very rapidly. A much cruder viewpoint says that the sub-set is determined just by its smallest and greatest members. In this case the type would be

min-evenint .. maxint/2

and this is the type chosen for Ten15. This is an arbitrary notion of type, but it has the advantage that it can be computed and that it is useful for all the purposes outlined above. All type systems in use are arbitrary in some such way. For the purposes of Ten15 a type system must of necessity discard information, and there is no uniquely satisfactory way of discarding it, nor need there be.

The Ten15 notion of type, then, is an arbitrary but carefully chosen collection of sets which is such that the type of the result of each of the machine operations can be calculated if the types of its parameters are known.

An area where programming languages have differed is in type declaration. There appear to be three different ideas involved. First, a new type could be defined to be equal to some composite type purely as a shorthand way of referring to that type. Second, a new type could be defined as being a different kind of thing from any present type, and represented by some existing composite type. This is really part of the notion of abstract data type. Third, a type could be defined as being equal to a type expression as part of the definition of a circular type, that is, the intention is to solve the type equations. Pascal mixed up the functions of the first two reasons, and made similar type declarations in different places in the text produce different types. The intention was probably to achieve the effect of abstract data types, but this was not made explicit. The effect in fact was to tie these type declarations to the text in a way which made it difficult to make them global. Algol68 used type declarations for the first and third purpose. This meant that it was necessary to recognise what was going on if one wanted to see which were the circular types. Ten15 provides abstract data types and circularity as separate essential constructions; renaming is available, but merely a convenience.

The type system is built up from basic types, from type constructors which form types from given types, and in more complex ways such as polymorphism and abstract data types. In various places in the basic types, pragmatic choices have to be made. This is especially true about floating point numbers, since, even with the advent of the IEEE standard, there is a great variety in floating point implementations on machines. A choice which made great difficulty for some common machine would be unfortunate, so compromise forces a less-than-ideal solution. Similar remarks apply, though

less strongly, to integers, where different word lengths and conventions about signed and unsigned arithmetic apply. Apart from these two major areas, it has been possible to make satisfactory decisions, without much constraint arising from the nature of existing machines.

9.2.1 Basic types

Ten15 has a large but finite set of basic types. This is not a necessity, as similar arguments could have led to an infinite but countable set of basic types.

The simplest basic type is *Void*, a type which has exactly one value. This being so, no bits are needed to represent it. The name *Void* for the set is a clear misnomer, but it has been traditional since the days of Algol68 and has been retained in Ten15. In effect, this is the value delivered by an operation which has no genuine value to deliver, because it is an operation performed for its side-effects. Thus assignment delivers a *Void* value.

Other simple types are *Bottom* and *Top*. *Bottom* is the empty set, which can only be the result of an operation if control cannot reach that point, and *Top* is the set of all values, which is not delivered by any operation, but can arise in error situations. The names are chosen because of the position of these sets in a lattice of types.

Integer types force the consideration of the word lengths of practical machines. If arbitrary integer ranges were allowed, some constructions might be excessively slow. These considerations dictate a compromise, which may be uncomfortable to the purist, but which has worked without much awkwardness in practice, no doubt because of the similarity of many current machines. Ten15 has chosen a number of preferred lengths, corresponding to one bit, eight bits, 32 bits and 64 bits, and permits contiguous subsets of the corresponding numbers as integer types. Normal arithmetic operations are defined on these types. With this scheme a type might be (64 bits, -5 .. 63) or (8 bits, 10 .. 10). The latter type contains only the one value and is the type of a suitable constant value in a piece of Ten15. The subset information is used to guarantee that operations are permissible without run-time checks; for example indexing a vector by means of a subscript known to lie within its domain need not be checked. This means that overflow must always produce an exception, since an unchecked overflowed value might not lie in the correct range.

For *Real* types, again something of a misnomer, floating point would be better, Ten15 works in terms of numbers of bits, in this case significant bits in the mantissa and exponent. A *Real* type is specified just by these two parameters, and questions of accuracy and range are not dealt with in the type system. The usual arithmetic is defined.

There are a small number of cases where run-time checks have to be performed because, in some cases, they cannot be made at compile time from the particular types given. The important ones are conventional:

overflow, division by zero, and index check. These checks can usually be implemented directly in terms of hardware constructions. The mechanism is described later, but it involves a type, *Exception*, which gives information about such errors back to controlling procedures. Though this control is provided in Ten15, it is not intended for use as a control mechanism by normal users since it seems to be bad practice, though it is required by Ada. Other safer mechanisms are provided in Ten15.

There are other minor basic types which are not discussed here, including the type of type values itself. The only other major basic type is *Typed*, which is described later.

9.2.2 Type constructors

Given any type, X, a type of vectors of such values can be created. On this type the main operations are the creation of the vector, indexing and obtaining the number of elements. So *Vec* is a way of making a new type from a given type and there are many such type constructors.

The type constructors are not confined to having only type parameters: *Vec*, for example, takes a type parameter, and also a boolean parameter which says whether values of the type are read-only. It also takes a further type parameter, the type of the size of the vector, so the full vector type has the form

Vec(type of item, read only, type of size)

Vectors are indexed by positive integers. A compiler, say for Pascal, wishing to translate an index which is an enumerated type, will use the representation of that enumerated type. If the type of the size of a vector is given as (32 bits, 5 .. 5), for example, then the vector will have exactly 5 elements. This will enable the translator to omit index checking if the type of the index is appropriate. Indexing the vector gives a reference to a *type of item*, and this reference is read-only if the vector is read-only.

It is frequently necessary to consider whether one set of values is a subset ($\subseteq$) of another. This is affected by the type constructor. For example, if $A \subseteq B$, the relation between *Vec* A and *Vec* B can be considered. It is possible to put values into the vector by assignment to a reference obtained by indexing, and get values out by de-referencing. If *Vec* A were a subset of *Vec* B, then every operation which can be done to a *Vec* B would be possible on a *Vec* A. However, a value of type B can be put into a *Vec* B but cannot necessarily be put into a *Vec* A, since some B's may not be A's. So *Vec* A is not a subset of *Vec* B. Conversely, if *Vec* B were a subset of *Vec* A, the values obtained by applying an operation to a *Vec* B would be a subset of the values obtained by applying the same operation to a *Vec* A. But the values that can be extracted from a *Vec* B belong to B, and those from a *Vec* A to A, and B is not necessarily a subset of A. So even though $A \subseteq B$ neither *Vec* A nor *Vec* B is a subset of the other.

This holds because the vector types could be both assigned to and de-referenced, which is true if neither is read-only. But suppose that both vector types are read-only. In this case it is true that *Vec A* is a subset of *Vec B*.

Values of type *Ref X*, are references to values of type *X*. Again there is a read-only property. De-referencing is defined on all references, and assignment is defined if the reference is not read-only.

Values of type *Ptr X* have de-reference and assignment defined, but are not able to point inside other structures. They can be implemented by packing the value of type *X* into a new block and giving the address of that. Whereas references might be to parts of such a block, pointers have to be to the whole of it. Pointers are only provided because they can be implemented in less data space than references; efficiency rather than necessity is the rationale.

The type *Struct*$(A, B \ldots N)$, where $A, B \ldots N$ are any number of types, is the Cartesian product of its parameters. The operations are tupling, which creates a structure, and selection, which selects a field from a structure. Here $P \subseteq Q$ implies that *Struct*$(P, B \ldots N) \subseteq$ *Struct*$(Q, B \ldots N)$ and similarly for all fields.

There are also union types, *Union*$(A, B \ldots N)$. This is disjoint union, not set union. It can be implemented as sets of pairs of numbers and values.

$$(1, a_1), (1, a_2), (1, a_3) \ldots (2, b_1), \ldots (n, n_1) \ldots$$

$P \subseteq Q$ implies *Union*$(P, B \ldots N) \subseteq$ *Union*$(Q, B \ldots N)$.

Values belonging to *Proc*(X, Y) are procedures, expecting a parameter in X and delivering a result in Y. A *Proc*$(1 .. 10, Y)$ can be used everywhere where a *Proc*$(2 .. 5, Y)$ is needed, since the first proc is applicable to all $2 .. 5$ values. So if $A \subseteq B$, then *Proc*$(B, Y) \subseteq$ *Proc*(A, Y). This is an illustration of the usual Galois relation between sets of values and sets of procedures. In Ten15, a procedure has just one parameter, so the *Struct* construction is used to group them if more are needed.

A special type construct, *Unique X*, is discussed in a separate section. There are other minor type constructors for main-store values, but the major type constructors which remain are concerned with backing store and with network values. *Remote X* is the type of any value of type *X* in another computer on the network. Full details are beyond the scope of this chapter, but are discussed in [Foster/Currie1986].

In Ten15, backing store information is organized into data stores, which are identified by values of basic type *DataStore*. A computer may have access to any number of data stores. In Ten15 operations are provided to manipulate these explicitly, though of course the data store operations can be made invisible to users by software built on top of these primitives. The types involved are *Persistent X* and *Persistent-Variable X* [Currie/Foster/Core1987, Atkinson/Morrison1985]. Any type of value can be written to data stores and the result of writing a value of type *X* is a *Persistent X*. The

operation to read this value back produces a copy of the original value, of exactly the same structure: that is, common pointers and references remain common. Writing a value in this way puts it in a new area in the data store as an atomic operation. In order to be able to change information in a data store a persistent variable has to be used. The operations on this are assignment and de-referencing, just as in main store. Ten15 provides facilities for the atomic update of a number of persistent variables within one data store. The more difficult problem of atomic update of data distributed between different computers on the network can be solved, but the primitives for this are still under development.

9.2.3 Circular types

Consider the type of conventional linear lists of values of type A. Such a list might be either null or a pointer to a structure containing an A and also another list:

List = Union(Void, Ptr Struct(A, List))

This equation is not just a renaming for convenience but has in some sense to be solved in order to obtain the actual type. Exactly the same techniques used elsewhere in this chapter to explain loop constructs in programming can be used here to give a meaning to the solution of this equation. An equation-solving operator, Y, is introduced, so that *List* can be defined in a way which is a renaming:

List = Y λt. Union(Void, Ptr Struct(A, t))

The Y operator, least fixed point or equation solution, can be extended to multiple equations in a direct manner. It can be shown that the equations can be solved, one at a time, in any order, giving the same result. This interchange theorem also applies to program semantics.

9.2.4 Polymorphism

Ten15 supports both conventional kinds of polymorphism [Reynolds1985]. If lists are as defined above, a procedure, *map*, could be defined with two parameters, one a *List* and the other a *Proc*(A, *Void*)—taking an A parameter and delivering nothing—which applies the procedure to each member of the list:

map ∈ Proc(Struct(List, Proc(A, Void)), Void)

The body of the procedure *map* would be the same no matter what the type A was. It is quite practical to implement *map* so that the same code will suffice for all A. The implication is that the value *map* is a member of all the types obtained by putting a particular type in for A. Hence it is in the

intersection of all such types. This type will be written as

$$\cap\, \lambda u.\ Proc(Struct(Y\ \lambda t.\ Union(\ Void,\ Ptr\ Struct(u,\ t)),\ Proc(u,\ Void)),\ Void)$$

meaning the intersection of all types arising by substitution of any type for u in the lambda expression.

The procedure *map* is polymorphic in the sense sometimes referred to as ∀-polymorphism, since the value lies in *all* the types of the given form.

It is important to be clear about the difference between the type of *map* looked at from the outside, where it lies in all suitable types, and the type of the parameters of *map* looked at from the inside, that is how the parameters appear in the body of *map*. When the body is being compiled the parameters are known to lie in a particular instance of their form. During the whole of the body it is known that the same u is involved, although it is not known what actual type u is. This is a dual kind of polymorphism, often called ∃-polymorphism. The value lies in a type which is some instance of the given form.

As another example, consider the operation de-reference and the set of values to which it is applicable. Can this set be described as a type? It consists of all values of types of the form *Ref X* for some X. That is to say they are in the set union of all these types. By analogy this type is $\cup\lambda u.\ Ref\ u$. When de-reference is applied to a value of this type it is known to be a reference, but the type of the contained value does not have to be known in order to see that the application is permitted. This kind of polymorphism has been described as being what is involved in abstract data types [Mitchell/ Plotkin1985], but in Ten15 it is used much more widely.

A further very important use is discussed in the section on the type *Unique X*.

9.2.5 Abstract data types

Abstract data types have been included in Ten15, though it would be possible to argue that they were more properly constructions of the languages that might be translated into Ten15 than of Ten15 itself. On balance it seemed better to provide a standard mechanism, following the argument that Ten15 must be a wide-spectrum machine in order to permit transformations.

It is possible to define both new basic abstract types and new abstract type constructors. In all cases the mechanism is to specify the representing type and the defining operators. The change from abstract type to its representation is itself protected from unauthorized access by the type checking, in the same way as other forms of access control.

9.2.6 The type *Typed*

Ten15 itself is strongly typed, but so far in this chapter no mechanism has been described by which programs can be written which themselves check types, at least not in a way which makes sure that they cannot err. Nor has a way been given of writing programs which handle values of types not known at compile time. Such programs are needed; for example, a command line interpreter will have to handle values of types determined by the user, and these must not be limited to a fixed finite set of types by using the Union construction, which is the only mechanism so far described which is at all helpful. Furthermore, the command interpreter must check types before it applies procedures to their parameters, and this check must be guaranteed by the type system.

Ten15 provides the single basic type *Typed*. This can be thought of as a value and its type wrapped up together. There is an operation to extract the type from a *Typed*, but this cannot be used to do guaranteed type checking, since the association between this type and the original value is lost. The typed value is something like the union of an infinite number of types, and so it would be possible to consider a *case* construction to decompose the values, one level at a time, without loss of strong typing. After experimenting with this approach it became clear that it was too difficult to implement effectively. Ten15 in fact implements operations on typed values which are derived from the corresponding ordinary operations.

For example, there is an operation, *Call*, which has two parameters, a procedure, f, of type $Proc(A, B)$ and a value, v, of type A. *Call* applies the procedure to the other parameter and delivers a value, $f(v)$, of type B. From this an operation is derived which takes two *Typed* values, checks that the actual types correspond, applies the procedure, and delivers the resulting values as a *Typed*. This is just like interpreting the *Call* operation, the data being everywhere typed. It is a heavyweight operation, so other methods should be used whenever possible.

A typical use of *Call* is in a command line interpreter. A *Typed* value has been obtained by looking up a name in a dictionary; the user thinks it is a procedure and has also provided what seem to be suitable parameters; again these are *Typed* values. Now the user asks for the procedure to be applied to the parameters. The command line interpreter is an ordinary program which has been produced by translating Ten15 into machine code. In order to make sure that the procedure can only be applied to parameters of the correct type the command line interpreter uses the typed version of the *Call* operation. The translation of this operation into machine code contains instructions which check that the actual typed values supplied to it do indeed match in the correct way. There is no way of applying the procedure to its arguments without performing this check. Clearly in this case the check has to be dynamic, since it was not known what procedure value might be fetched.

9.2.7 The type *Unique X*

Every designer of PSEs has found that many uses can be made of a way of constructing tags such that each new tag created is guaranteed to be different from every other tag on the current machine and every other machine. In effect this can be implemented as a combination of the identity of the current machine and the date and time. In Ten15 it has been found that this idea is especially useful if these tags have a type associated with them, so *Unique X* is a type constructor with one type parameter.

As an illustration of this Ten15 use, consider a structure of the following type:

$$\cup\ \lambda t.\ \mathit{Struct(Unique\ t,\ Ptr\ t)}$$

Suppose there is a tag which is a *Unique Real*. This can be compared with the *Unique* value in the structure, since equality can be tested between *Unique* values of whatever type. If they are equal, that in the structure must be a *Unique Real*, and so the associated pointer must be a pointer to a *Real*. Ten15 provides an operation (of the kind called assertion in Ten15 and described later) with two arguments, a *Unique* value and a pair of this sort, which performs this test and delivers the associated pointer value, now with known type. Given a vector of such pairs, in which each pair contained in the vector can be a different instance of the form, and given a *Unique*, the vector can be scanned, using this operation, to look for an associated value and to obtain it with known, guaranteed type. This is much more efficient than using a *Typed* value for the same purpose.

This is an operation of wide utility and an interesting use of the ∃-polymorphic types. The pair is something like a Union, but the decision about which types to unite has been delayed. It has proved particularly useful in writing a loader, which would otherwise have had to use the much slower interpreted *Typed* operations in a place where speed is very important.

9.3 Features of the Ten15 machine

In many respects Ten15 is conventional, as indeed it must be to act as an easy target for conventional languages. It provides such constructs as case expressions, conditionals, loops of various sorts, integer and real arithmetic, boolean operations, string handling, vectors and arrays. This section will take these for granted and concentrate on the less usual features.

Ten15 is defined algebraically; complex pieces of program are built out of less-complex ones by means of composition constructs. It is essentially tree-like. As an example a conditional expression is built out of an expression which delivers a boolean and two expressions which deliver values of types which have a least upper bound (not *Top*). Each of the expressions

below, and the whole, are pieces of Ten15 program:

if(boolean expression, expression1, expression2)

The expressions just considered are called *Loads* in Ten15, since a technical term is needed. The name is chosen because *Loads* produce values when evaluated in some state of the Ten15 machine. They may also produce a change in the state, that is a side-effect. Note that these are items in the Ten15 algebra, that is fragments of program for a Ten15 machine.

A unary operator is a piece of Ten15 which takes in a value and from it produces a value, whereas a *Load* produces its result value from nothing. A conventional basic unary operator, such as *Not*, can be applied to a *Load*, using the construct *operate1*.

operate1(Not, expression)

The result of this is itself a *Load*.

The standard unary operators are examples of *Unary-Operator*, but operators do not need to be basic and can also be built up by means of Ten15 constructs. One construct in particular, *make-op*, builds a *Unary-Operator* from a *Load*, *exp1*, and an *Identifier*, *id*. Operating with such an operator on a *Load*, *exp2*, identifies the value produced by *exp2* with the given *id* and evaluates *exp1*, using that value wherever *exp1* contains *id*.

9.3.1 Assertions and solve

Ten15 has to have both high-level and low-level features. Among the low-level features are some which correspond roughly to labels. This is an inevitable consequence of the aim of providing a wide-spectrum machine and being able to transform programs sufficiently close to conventional machines to achieve efficiency. But unrestricted labels and *gotos* lead to programs which can be difficult to analyse. The Ten15 mechanisms for achieving these effects are assertions and the *solve* expression.

Ten15 takes the viewpoint that a collection of labelled expressions is in fact a set of simultaneous equations, in which the labels play the role of the variables being solved for, and may occur in the expressions. Solution is interpreted with a least fixed point meaning. This being so, the labels would be formal *Loads*, for which the equations could be solved to produce a collection of actual *Loads*. The *goto labels* would correspond to the uses of the formals in the equations. In practice, real machines jump with values and unary operators expect values so the labels are formal *Unary-Operators* rather than *Loads*. The solution of the simultaneous equations produces a collection of actual *Unary-Operators*. This, among other advantages, gives a much better loop construction. The construct *solve* takes number of pairs of formal operators, the labels, and actual operators, and solves the equations.

Within the context of a *solve* construction a straightforward jump is represented by a construct which takes a label and builds a *Unary-Operator*,

g. When *g* is applied to a value it will jump with that value to the label. Control never reaches the point immediately after the jump, so the value there is of type *Bottom*.

High-level constructions, such as *case union*, are expressible in terms of lower-level constructions. Clearly *case union* works by examining the union value to see which of the possible members is actually present and jumping to the appropriate place. Note how, in this construction, jumping with a value is right. The branch of the *case union* must be arrived at with the component value present and ready to be identified.

Conventional programming languages, because they use boolean values to control conditionals and loops, make it difficult to retain type information which should be available. Consider as an example the conditional

IF x < 0 THEN -x ELSE x

It is clear that if x is in $-10..10$ then the result will be in $0..10$. Not many compilers would notice this fact. Going through the boolean value has made it difficult to obtain the type information about x in the two branches. Ten15 uses *Assertions* to generalize the notion of conditionals while simultaneously making type information more explicit.

The *case union* shows how it can be done. In place of the conditional, control must split into the two possibilities, carrying into each branch a value of the appropriate type which can be identified in the branch. Making up some syntax by way of illustration:

```
IS x < 0 ?
  IF SO  t: -t
  IF NOT u: u
```

If $x < 0$ then take the first branch, identifying x, now known to be negative as t; if not take the second branch identifying x, now known to be non-negative, as u. If x is in $-10..10$ then t is known to be in $-10..-1$ and the result of the first branch is in $1..10$. In the other branch u is known to be in $0..10$ and so is the result of that branch. Hence the result of the whole construction is the least upper bound of $1..10$ and $0..10$, which is $0..10$.

The two branches are unary operators, created by *make-op*. The test is an *Assertion*, and when combined with the two *Unary-Operators* and the two argument *Loads*—for x and zero—it yields a *Load*.

It is interesting to contrast this approach with that of "continuations". In Ten15 the program, which might be unbounded but countable, is created by solving the program equations before any question of evaluation arises.

9.3.2 Procedures and ions

Ten15 supports true procedure values with an unbounded number of binding

times for non-locals [Landin1964]. These partly bound values have been called *ions*.

The Ten15 procedure values are just the conventional closures of Landin, except that instead of having only one bind time for non-locals, many bind times have been allowed for. Though one bind time is logically sufficient, having many allows for much greater efficiency in some important cases. Ten15 calls the partly bound objects ions.

Consider a procedure created inside another one and treated as a true value, so that the "display" implementation will not work. It is normal that the internal procedure will have some non-locals which are non-locals of the external procedure. Conventionally this would cause us to bind these values as non-locals of the external procedure, and then copy them into the non-locals of the internal procedure. Very often this will be the only use of these values in the external procedure. In this case it is clearly better to bind them as the first of two sets of non-locals of a constant ion, and to give this partly bound ion as a non-local to the external procedure. This then just copies out those of its locals which form the second set of non-locals to the ion to give the required internal procedure.

9.3.3 Exceptions

When an operation fails a run-time check, as for example if an arithmetic operation produces an overflow, an exception is produced. This exception is treated as a value, and every operation but one which has an exception value as a parameter is deemed to give an exception result. This would fail all the *Call* operations which had called procedures leading to this situation. If this were all, control would rapidly fall out of the program, and since the operating system itself is also in Ten15, the whole machine would stop. However there is an assertion, called *trapply*, which can be used to prevent this. This assertion takes *Unary-Operators*, the second of which expects a value of type *Exception*. From these is built a binary operator which takes a procedure and an appropriate parameter for that procedure. The effect of the assertion is to apply that procedure to its parameter in just the same way that the ordinary *Call* operator does. If the procedure when run produces an exception, then the assertion uses the second operator, giving it the value of type *Exception* produced. This value contains information sufficient to produce diagnostics if necessary. If the procedure terminates normally, the first operator is used, so it must expect a value of the type delivered by the procedure. The whole delivers a value of type given by the least upper bound of the types produced by the operators.

Normally when an exception value is delivered, action will be taken to clean up the state of the machine, stuck semaphores will be cleared and so on. It is not expected that this mechanism should be used in ordinary users programs, but since users can write programs which are like operating systems, these facilities have to be available.

9.4 The formal method

Ten15 is defined entirely by means of intrinsic equations which say which programs are equivalent. There is space here only for a brief exposition and discussion of the method.

A system will be defined which is closely related to a conventional many-sorted algebra [Goguen 1976]. The definition starts with a finite number of "sorts" of which *Load*, *Operator* and *Assertion* are examples, and then introduces a finite number of constructs, each of which takes a number of parameters from particular sort and produces a result in a given sort. For example,

$$operate1 : Unary\text{-}Operator, Load \rightarrow Load$$

Some of these have no parameters, in which case they denote constants in the result sort. For example,

$$load\text{-}void : () \rightarrow Load$$

which will be written as

$$load\text{-}void : Load.$$

The sorts and the constructs together form a "signature".

Some of the sorts are designated as semi-lattices, in which case they contain the constructs, least upper bound, written $\cup$, and bottom, written $\perp$. Least upper bound is associative, commutative and idempotent, and has bottom as its identity. If the sort is L, then

$$\cup : L, L \rightarrow L$$
$$\perp : L$$

Given these definitions a partial order, $\leq$, can be defined by saying that $a \leq b$ is equivalent to $a \cup b = b$.

Let there be, for each of the sorts, a countable set of variables. Now define inductively a set of terms. With each term will be defined its sort, and its set of free variables. For each sort, S, S_0 consists of all the constructs with no parameters which have S as result, and also all the variables of sort S. The constructs have no free variables; the variables have themselves as their only free variable.

For each n, S_n contains the following constituents and no others. First, it contains all the expressions, $f(t_1, t_2, \ldots t_k)$, where the terms t_i have the sorts required as parameters by the construct f. The t_i lie in sets indexed by $m < n$, and at least one of them lies in a set indexed by $n - 1$. The result sort of f is S. The free variables of this term are the union of the free variables of the t_i. Second, if S is a semi-lattice sort, S_n contains all the terms $Y\lambda x.\, t$, where x is a variable in S and t is a term in S_{n-1}. The part of this term after

the Y is called the controlled λ-expression. The free variables of this term are the free variables of t less x. Y is to provide a method of solving equations.

Now the set of terms in S is defined to be the union of all the S_n. The terms in S which have no free variable are called the ground terms. The carrier set of the algebra for each sort consists of the ground terms belonging to that sort. This construction is very like a conventional word algebra, differing only in the introduction of the Y terms, and it will be called a word algebra in this chapter. The word algebra is determined entirely by the signature and by which sorts are semi-lattices, for example one can speak of the Ten15 word algebra.

Congruences are defined in the usual way to be equivalence relations, one for each sort, which have the property that if in $f(t_1, \ldots t_n)$ and $f(u_1 \ldots u_n)$ the parameters are pairwise equivalent, then so are the terms. The quotient of the carrier set by a congruence is also defined conventionally.

A word algebra and its quotients form a class of algebras. It is a class of algebras of this nature which will be used to define and study Ten15. A major topic, therefore, is how to specify congruences.

One common technique for specifying a congruence is to give a set of "laws" which specify which pairs of values are to be equal. If the laws are of an appropriate kind there will always be a smallest congruence which satisfies them. The signature, the semi-lattices and the laws then define an algebra.

The Ten15 word algebra defines the set of Ten15 programs which are well-formed, but nothing is known about their meaning. One could perhaps call this the abstract syntax of Ten15. As more laws are specified, the quotient by the smallest congruence approaches more closely to the required semantics of Ten15, which could be called the semantic congruence. This is not to indicate that this process is intended to be a limiting one; all of the necessary laws will be given, though not in this chapter. But it is valuable to look at particular programs in more or less detail, that is with more or less laws, in order to analyse them.

As an illustration of this, consider the simple but useful task of examining a piece of Ten15 and listing the identifiers which it uses. Clearly, making a systematic change to the identifiers would not affect the meaning of a Ten15 program; this would yield an equal program in the semantic congruence. Therefore programs which are the same except for the identifiers used must lie in the same congruence class of the semantic congruence. So one cannot ask about this congruence class "what identifiers does it use?". This question cannot be asked or answered in the full algebra. But in the word algebra, it is a possible question.

It is possible to define the usual idea of homomorphism mappings between quotients of the word algebra. The only extension consists of defining the mapping of the Y terms, to give the corresponding terms in the image. Many analytic properties of programs may be obtained by evaluating homomorphisms, including for example the task of the previous paragraph.

9.4.1 Particular laws and Y

The form of laws needs to be chosen, especially the treatment of the Y terms. The simplest way of specifying laws is to give a number of pairs of terms of the same sort, $t_1 = t_2$. The interpretation of such a law is that equality is to hold for all systematic substitutions of the free variables contained in the terms by ground terms of the same sort as the variables.

Such a set of laws is not adequate to carry out the definition. The form of the laws can be extended by allowing conditional laws of the form

$$t_1 = t_2 \wedge t_3 = t_4 \ldots \rightarrow t_n = t_{n+1}$$

with the understanding that for every substitution for all the free variables by ground terms, if the terms on the left of the implication sign are equal as specified, then so is the term on the right. Even so, a finite number of such laws is not enough. A countable infinity is needed, with a finite way of expressing them.

In fact enough generality can be achieved by using special sets of variables to stand for the lambda expressions of each sort in laws of the above conditional form. These laws are first instantiated by lambda expressions for the special variables and then by ground substitutions for the ordinary variables. This can easily be shown to define a smallest congruence.

Of course, Y is intended as a least fixed point operator in the semi-lattice sorts, thus providing solutions for the equations which define loops and recursion [Landin1964]. Ten15 does not in fact use *least* fixed point as the basic definition. Let the variables for lambda expressions be Greek letters. The laws are

$$Y\alpha = \alpha(Y\alpha)$$
$$\alpha(z) \leq z \rightarrow Y\alpha \leq z$$

The first says that Y is a fixed point, the second that it is a lower bound to all values which are decreased by α (less than or equal). From these it is a consequence that Y gives the least fixed point of α and the least upper bound of the approximants, $\alpha^n(\perp)$. It is not the case that substituting the second law by a least upper bound law would be equivalent.

These laws have been chosen for Y because it is possible to prove an important interchange theorem which says that multiple equations can be solved one at a time in any order, rather in the fashion of Gaussian elimination for simultaneous equations, and the results will be equal no matter what order is chosen. Furthermore, the second law is a natural, finite way of expressing what is required.

9.5 Formal definition of Ten15

The purpose of this section is to illustrate the laws of Ten15 and to convince the reader that intrinsic equations are sufficient to define its semantics. It is

easy to see that more complex constructions such as *for* statements and *case union* can be defined by equations in terms of simpler ones. This section will be confined to showing that the same is true of declarations. It will omit many important areas, such as the definition of assignment and procedure values.

Some more of the constructs for building Ten15 and some more sorts will be needed. *Value* is the sort of the values which Ten15 programs manipulate. It is defined algebraically, just like all the other sorts.

The discussion will be confined to these constructs.

$$\begin{aligned} \textit{load-name} &: \textit{Identifier} \rightarrow \textit{Load} \\ \textit{load-value} &: \textit{Value} \rightarrow \textit{Load} \\ \textit{seq} &: \textit{Load}, \textit{Load} \rightarrow \textit{Load} \\ \textit{operate2} &: \textit{Operator}, \textit{Load}, \textit{Load} \rightarrow \textit{Load} \\ \textit{identity} &: \textit{Identifier}, \textit{Load}, \textit{Load} \rightarrow \textit{Load} \end{aligned}$$

The intuitive interpretations are as follows. The construct *load-name* produces as its result the value associated with the identifier by some governing declaration, and does not change the machine state. Likewise *load-value* is a piece of program which delivers the value parameter. This is therefore a denotation for a constant, and it is an important part of our equational definition mechanism, since it can be applied to any kind of value, reference, procedure or persistent value as well as integer.

A sequence is constructed from two *Loads*: it discards the value produced by the first parameter, and produces that given by the second. The sequence construction will only become important when assignment is introduced, and the first parameter might change the state of the machine, but the laws for sequences will be described. For example the associative law for sequences is

$$\textit{seq}(x, \textit{seq}(y, z)) = \textit{seq}(\textit{seq}(x, y), z)$$

For binary operators, *operate2* applies the binary operator to the two arguments.

The *identity*, a declaration, associates the value delivered by its first *Load* parameter with the *Identifier* while it processes the second *Load*. During this processing, wherever a *load-name* for the *Identifier* is used, the value produced from the first *Load* is intended. In order to be able to use identifiers, one must be able to determine whether or not they are equal. Notice that the method of definition enables us to state that two things are equal, but inequality is quite another matter. If inequality is to be used, then what is effectively an algorithm for determining it must be given. To be able to do this another sort, *Bool*, is needed. This contains two values, *true* and *false*, and no laws relating them. So they will be different in the smallest

congruence. Let

zero: () → Identifier
succ: Identifier → Identifier

and let there be no more laws for *Identifier*. Then the smallest congruence provides the other three of Peano's axioms and makes *Identifier* just the natural numbers. This is more convenient than defining *Identifier* as a string of characters.

The auxiliary construct *eqid* is introduced in order to define equality and inequality of *Identifiers*. The sorts and laws for *eqid* are

eqid : Identifier, Identifier → Bool
eqid(zero, zero) = true
eqid(succ(x), zero) = false
eqid(zero, succ(x)) = false
eqid(succ(x), succ(y)) = eqid(x, y)

Clearly, because *Identifiers* are just the natural numbers, *eqid* is completely defined by these laws for any pair of arguments.

Some of the laws for the constructs will be discussed, with comments on them to clarify their purpose. The aim is to show that declarations can be defined with equations alone. A piece of Ten15 which uses only identifiers, which are declared within it, will be said to have no free identifiers. The laws discussed in this section, when applied to a piece of Ten15 with no free identifiers, which only uses the constructs introduced above, will show that it is equal to *load-value*(v) and determine the value, v. The proof of this result is not difficult and is left to the reader. A proof can be based on showing that every expression except a *load-value* can be simplified by the equations, and that this process will terminate. In the following laws simplification is achieved by using the equalities to replace the left-hand side by the right-hand side.

The first law shows the obvious basic meaning of declaration:

identity(i, x, load-name(i)) = x

In the next law the inequality of identifiers is used:

eqid(i j) = false →
identity(i, load-value(v), load-name(j)) =
load-name(j)

The following law moves a sequence out of the definition part of a declaration:

identity(i, seq(x, y), z) = seq(x, identity(i, y, z))

The computation of x, which is being discarded since it is the first parameter of a *seq*, is only being performed for the sake of its side-effects. Accordingly, provided that it is carried out before the computation of y, it can be inside or outside the declaration.

The strategy is to reduce the definition part of an *identity* to a *load-value*, so that it can have no side-effects. Then other laws are used to move that definition into the constructions in the controlled part of the identity. For example, the following law moves a definition part, which has been reduced to a *load-value*, into a sequence:

identity(i, load-value(v), seq(x, y)) =
seq(identity(i, load-value(v), x),
identity(i, load-value(v), y))

If the definition part were not a *load-value* but an *operate2*, it would have to be reduced to use the above law. So not only declarations but also *operate* have to be removed while still making the term "simpler". For each basic binary operator there is a function, f, such that a law of the following form holds:

operate2(b, load-value(x), load-value(y))
= load-value(f(x, y))

In fact, this is not really adequate, since operators can change the state of the machine, and the laws which are relevant to state change have not been introduced.

Two laws enable sequences to be moved out of *operate*:

operate2(b, seq(x, y), z) = seq(x, operate2(b, y, z)
operate2(b, load-value(v), seq(x, y))
= seq(x, operate2(b, load-value(v), y))

These two laws are not symmetric, because x might have side-effects. Whether or not this is so, the first law holds, but in the second law the operand must be in the form of a *load-value* so that the move can be made. Clearly this pair of laws defines the order in which the arguments of operators are evaluated. A different form could have been used if the order of evaluation were to be undefined.

By the next law a declaration is moved into an *operate2*:

identity(i, load-value(v), operate2(bin, x, y)) =
operate2(bin, identity(i, load-value(v), x),
identity(i, load-value(v), y))

and by the following law into another declaration:

```
identity(i,
        load-value(v),
        identity(j, x, y))

= identity(i,
        load-value(v),
        identity(j,
                identity(i, load-value(v), x),
                y))
```

The above laws are sufficient to remove declarations and *operate2* from a piece of Ten15 with no free identifiers, which uses only the given constructs. Since operators affecting the state of the machine were not included, sequences played a rather irrelevant role.

This can be taken as a complete definition of the meaning of pieces of Ten15 which use the constructs discussed, even though this has been shown only for those with no free identifiers. For consider a piece of Ten15 which does use some free identifiers, and so does not necessarily reduce to a *load-value*. It could be incorporated in a set of declarations which define the spare identifiers as *load-values*. For each way of doing this, the resulting program can be completely reduced since the result now has no free identifiers. Clearly this can be interpreted as a definition of the piece of Ten15, since we know its effect in any environment.

Of course there are many complications when the amount of Ten15 that is described by laws is extended, especially when the effect of the *Y* operator is included. No such simple way exists for seeing that the whole definition is adequate.

9.6 Conclusions

Enough experience has been gained to see that many of the aims of Ten15 have been met. Compilers have been written producing Ten15, as have translators from Ten15 to Flex and to Vax. Programs translated in this way run with conventional efficiency. Programs written directly for the Ten15 machine and translated run in some cases with substantially greater efficiency, because constructions are available which cannot be utilized by the conventional languages. Various homomorphisms have been tried. The translator which produces Vax code was written as a homomorphism on the Ten15 word algebra.

Other aspects of Ten15 need more examination. In particular, the

question whether the definition of Ten15 is sufficiently convenient to admit useful proofs being carried out is still to be decided.

In some respects Ten15 is still incomplete. The most important area in which this is so is that of close-coupled parallelism.

References

[Atkinson/Morrison1985] M. P. Atkinson and R. Morrison, "Types, binding and parameters in a persistent environment", *Workshop on Persistence and Data Types*, Appin (1985).

[Currie/Foster1987] I. F. Currie and J. M. Foster, *The Varieties of Capability in Flex*, RSRE Memorandum 4042 (1987).

[Currie/Foster/Core1987] I. F. Currie, J. M. Foster and P. W. Core, "Ten 15: an abstract machine for portable environments", *Proc. 1st European Software Engineering Conference*, Strasbourg, pp. 149–159 (1987).

[Foster/Currie1986] J. M. Foster and I. F. Currie, *Remote Capabilities in Computer Networks*, RSRE Memorandum 3947 (1986).

[Foster/Currie/Edwards1982] J. M. Foster, I. F. Currie and P. W. Edwards, "Flex: a working computer with an architecture based on procedure values", *Proc. International Workshop on High-level-language Computer Architecture*, Fort Lauderdale (1982).

[Goguen1976] J. A. Goguen, J. W. Thatcher, E. G. Wagner and J. B. Wright, "An initial approach to the specification, correctness and implementation of abstract data types", *Current Trends in Programming Methodology* (1976).

[Hoare1978] C. A. R. Hoare, "Communicating sequential processes" *Comm. ACM*, Vol. 17, No. 8 (1978).

[Hoare1987] C. A. R. Hoare, I. J. Hayes, He Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey, B. A. Sufrin, "Laws of programming" *Comm. ACM*, Vol. 30, No. 8, pp. 672–686 (1987).

[Dijkstra1976] E. Dijkstra, *A Discipline of Programming*, Prentice Hall (1976).

[Landin1964] P. J. Landin, "The mechanical evaluation of expressions", *Computer Journal*, Vol. 6, No. 4, pp. 308–320 (1964).

[MacQueen/Plotkin/Sethi1986] D. MacQueen, G. Plotkin and R. Sethi, "An ideal model for recursive polymorphic types", *Information and Control*, Vol. 71, pp. 95–130 (1986).

[Milner1980] R. Milner, *A Calculus of Communicating Systems*, Springer (1980).

[Mitchell/Plotkin1985] J. C. Mitchell and G. Plotkin, "Abstract types have existential type", *12th ACM Symposium on Principles of Programming Languages*, New Orleans (1985).

[Reynolds1985] J. C. Reynolds, "Three approaches to type structure", *Proc. TAPSOFT*, Springer (1985).

[Scott1970] D. Scott, *Outline of a Mathematical Theory of Computation*, Tech. Monograph PRG-2, Programming Research Group, Oxford University (1970).

[Spector1982] A. Z. Spector, "Performing remote operations efficiently on a local computer network", *Comm. ACM*, Vol. 25, No. 1, pp. 39–59 (1982).

[Xerox1981] Xerox Corporation, *Courier: the Remote Procedure Call Protocol*, Xerox Report XSIS 038112 (1981).