Ten15: an abstract machine for portable environments

I.F. Currie, J.M. Foster, P.W.Core
Royal Signals and Radar Establishment
St Andrews Rd, Malvern
Worcestershire WR14 3PS
England

Summary:
Ten15 is an abstract machine which is defined algebraically with strong typing enforced throughout. The structure and operations of this machine are sufficiently rich to allow the efficient implementation of a general purpose program-support environment extending over networks. Since Ten15 is an abstraction of programming languages rather than hardware, it also serves as a target in the compilation of standard languages. As well as giving common addressing mechanisms, Ten15 provides a common system of types which is enforced throughout the system whether in user's programs, system utilities or any other tools, alleviating many of the interfacing difficulties encountered when independently written programs try to interact. Porting an environment based on Ten15 to a new machine consists of writing a translator of Ten15 programs for the new machine together with a relatively small part of the system kernel mainly concerned with peripheral drivers; a normal bootstrap gives the ported environment. The resulting environment is one in which the type system ensures the integrity of any data or program in it and is used as the basis to give both security and privacy. In addition the algebraic nature of the machine helps one to do formal reasoning about programs running in it.

1. Introduction

Strong typing is generally accepted to be a highly desirable property of a high order language; among other things, it improves programmer productivity, increases program portability, and gives one greater confidence in the integrity and correctness of the running program. Most modern languages are strongly typed so that one is reasonably sure of the structural integrity of programs written in them, at least as far as the particular typing model can describe one's data and program structure. This typing model delimits the class of program that one can write in the language. Stepping outside this class forces one to cheat the type system to some degree, usually by going outside the language into some system level which has a far more rudimentary notion of the types of objects. Indeed there is usually no useful correspondence between the types used by a programming language (like arrays,lists or procedures) and those used by the underlying system (like files or commands). This contributes handsomely to the problems encountered interfacing independently-written programs, even when they are written in the same language. Usually the interface is reduced to a least-common-denominator consisting of files of characters where all the typing information of the results of the programs is lost. One cannot use the information about the types of values gathered in compilation either to provide more efficient interfaces or to give a basis for ensuring structural integrity across *different* programs as well as within individual programs.

The extension of the use of a common type system to cover all the levels of the use of a computer system immediately eases the interfacing problem. For example, user programs, system utilities, commands in the command interface, general tools etc, could all be described as being of type procedure with some parameter and result types. The interfacing of any of these programs together reduces to normal procedure or function composition with the normal typing rules for procedure application. There is no need to reduce an intermediate result to a character file or stream, thus losing all the structure of the data, and necessitating restructuring and re-validation before it can be used in further steps of the computation.

A common type system is only one facet of the interfacing problem, albeit an important one. A even more important one is having common methods of accessing data. A common "address-space" (as well as a common type system) in which the objects manipulated can be addressed uniformly reduces the interfacing problem by an order of magnitude. The particular methods of accessing these objects, together with the operations which one can apply to them, are determined precisely by on the type of the object. Thus we know that one can access the integer elements of an object of type vector of integer by indexing in a perfectly standard manner throughout the system. Furthermore, this is totally independent of how or where the vector was created; it could be the result of a system command, user program or even a value found in the diagnosis of a failed program. This unanticipated use of arbitrary values (including those which are, in fact, program) is highly important in any support environment; one seldom knows *a priori* just exactly what tools, programs and diagnostic aids one requires at any time. If a value is bound into particular address-space (eg a failed program) so that one cannot extract it in this unanticipated way, then the only tools that one can apply to it are those those which were put into the address space when the program was created. These tools could range from nothing to some fixed program to display some kinds of values; they seldom include the possibility of applying some program of one's own to the value.

Our approach is to define (algebraically) an abstract machine called Ten15. The operations in Ten15 follow strong-typing rules so that the correctness of the application of an operator (say) depends only on the types of its operands. Much of the motivation for inventing Ten15 was derived from its progenitor, the RSRE Flex system [Currie 81,82,85, Foster 82] which used firmware, rather than type structure, to enforce the correctness of the application of operations. The type system of Ten15 is sufficiently rich to usefully describe all the operations required for a general purpose program support environment (PSE) extending over a network; this, of course, covers a large proportion of the spectrum of computing. Any program running in an implementation of the Ten15 machine on some host is fully type checked by a trusted Ten15 translator. The type system,and the way that operations are defined within it, is such that one can ensure the integrity of storage allocation and garbage collection in mainstore as well filestore and across networks. In addition, it forms the basis of very natural ways to implement privacy between different users or between different areas of concern. Both the Flex system and Ten15 implementations are very similar in these respects, differing only in the methods of ensuring that crucial type rules are not broken. However, Ten15 is not an abstraction of the Flex machine (or any other concrete machine) but rather an abstraction of the concepts in programming languages in general. Thus Ten15 can be a target of a compiler for any standard programming language; this compiler need not be trusted since the Ten15 rules will be checked by the Ten15 translator.

The remainder of this paper is a sketch of the definition Ten15 with some indications of how it is implemented and how the higher facilities of a portable program support environment are built upon it.

2. Types and representations of Ten15 values

2.1 General

The types of values in Ten15 are form a lattice under the relation "can be coerced to" giving the automatic type changing rules. The form of this lattice is sketched in Figure 1. The lattice is designed so that an automatic type change of a value can only occcur where there is no change in the representation of the value; the Ten15 translator does not need to produce any extra code for such a coercion. A large proportion of the types and their corresponding operations in Ten15 have familiar analogues in programming languages and need little detailed description. The other less familiar types and operations are described below.

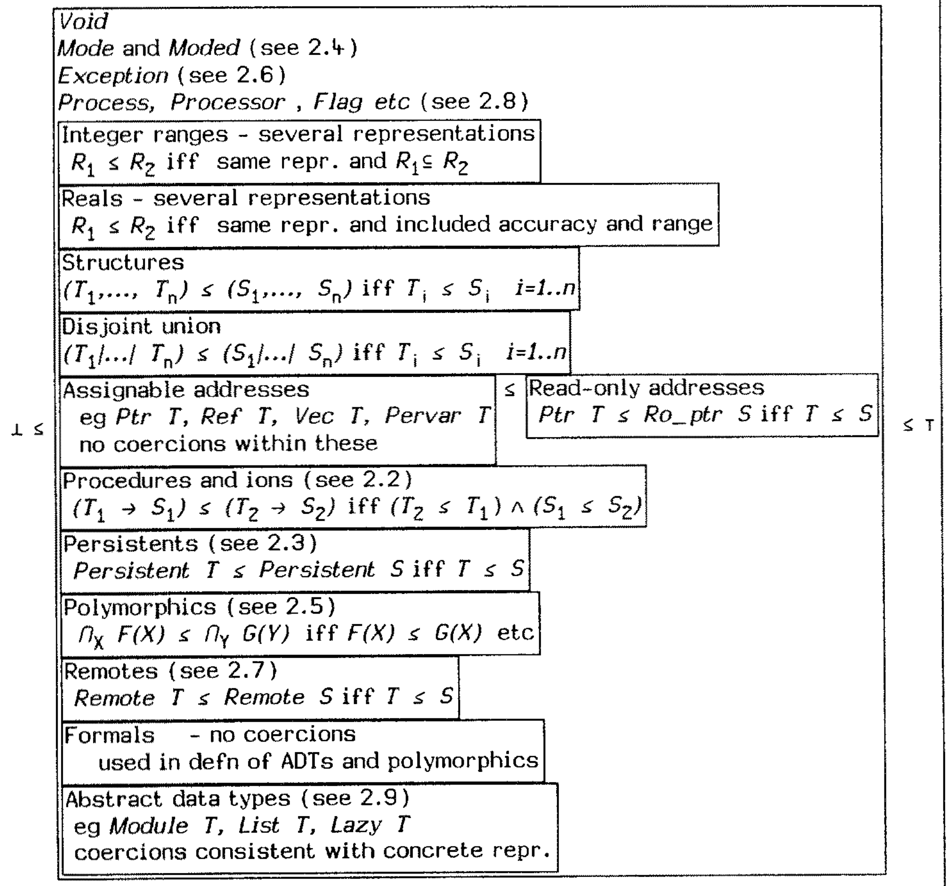
Void
Mode and *Moded* (see 2.4)
Exception (see 2.6)
Process, Processor , *Flag etc* (see 2.8)
Integer ranges - several representations
$R_1 \le R_2$ iff same repr. and $R_1 \subseteq R_2$
Reals - several representations
$R_1 \le R_2$ iff same repr. and included accuracy and range
Structures
$(T_1,\ldots, T_n) \le (S_1,\ldots, S_n)$ iff $T_i \le S_i$ $i=1..n$
Disjoint union
$(T_1/\ldots/ T_n) \le (S_1/\ldots/ S_n)$ iff $T_i \le S_i$ $i=1..n$
Assignable addresses
eg *Ptr T, Ref T, Vec T, Pervar T*
no coercions within these
$\le$
Read-only addresses
Ptr T $\le$ *Ro_ptr S* iff $T \le S$
$\perp \le$
$\le \top$
Procedures and ions (see 2.2)
$(T_1 \rightarrow S_1) \le (T_2 \rightarrow S_2)$ iff $(T_2 \le T_1) \wedge (S_1 \le S_2)$
Persistents (see 2.3)
Persistent T $\le$ *Persistent S* iff $T \le S$
Polymorphics (see 2.5)
$\Pi_X\ F(X) \le \Pi_Y\ G(Y)$ iff $F(X) \le G(X)$ etc
Remotes (see 2.7)
Remote T $\le$ *Remote S* iff $T \le S$
Formals - no coercions
used in defn of ADTs and polymorphics
Abstract data types (see 2.9)
eg *Module T, List T, Lazy T*
coercions consistent with concrete repr.

Figure 1 - The type lattice of Ten15

2.2 Types for program values

As befits a system intended for program support, Ten15 values include several kinds of program values. The most familiar are procedures which are first class objects in Ten15; they can be assigned, held in data structures or delivered from other procedures just like any other value. The representation of a procedure in mainstore is basically pointers to blocks containing things like code and non-locals. Its type is given by its parameter and result types; for example:

isin: ([1..100000] → Bool) (1)

is a procedure with an integer parameter delivering a boolean. A call of a procedure is just the most dynamic way of binding new values (the parameters) to a piece of program. Other binding times are possible and Ten15 permits one to bind values to other kinds of program values in a general fashion. The most basic program value in Ten15 is a "nucleus" consisting only of the code of the procedure. This code will, in general, access some free variables and its correct operation will require some values to be bound as non-locals to replace these free variables. These values can be bound in stages producing "ions", each different ion having additional non-local values until all are bound giving a runnable procedure. The types of the non-local values (and the order in which they will be bound) are all included in the type of the nucleus to ensure that only the correct kind of values are operated on by the code of the procedure. For example, the nucleus of the procedure *isin* in (1) might be an algorithm expressed roughly as:

$\lambda x.\ v[f(x)]$

with type:

([1..100000] → [1..10]) ~ Vec Bool ~ ([1..100000] → Bool)

by first binding *f* (a procedure of type *([1..100000]→[1..10])*) to give an ion of type:

Vec Bool ~ ([1..100000] → Bool)

and then, sometime later, binding *v:Vec Bool* to give *isin*. Note that the value *isin* is independent of any scope restrictions; its existence does not depend on any contexts of *f* or *v*. Indeed,it is easy to arrange matters so that a call of *isin* becomes the only way to access *f* or *v*.

The generalisation of this idea of hiding values behind the procedure interface is the basis for implementation of the privacy and security aspects of the Ten15 PSE. For example, the ion corresponding to a user's log-in procedure has a type something like:

(Dictionary, Password, ...) ~ (Void → Void)

where *Dictionary* and *Password* are some persistent data structures giving name look-up and the password mechanism. Once the actual values of these data-structures have been bound to this ion to produce the procedure, then the only way to access these data structures could be a call of the procedure. Of course, if this is the case the user had better not forget his password!

2.3 Types for database values

Database values are clearly required to allow one to keep data on persistent storage media. These are represented by values which are effectively pointers to blocks on the particular medium and whose type is made using a *Persistent* constructor. For example, a page of text lines might have a representation whose type is:

Persistent Vec Vec Char

This value would have been the result of applying a Ten15 operator (persist) to a value of type *Vec Vec Char* which could then be retrieved at any time by applying an un_persist operator to the persistent value. It is often very useful to have persistent data structures which contain other persistent values thus producing tree-like structures (strictly speaking, acyclic graphs) in a database. For example, a better representation of a text file might be one which allows sub-files with circular type, *Txt*, where:

Txt = Persistent Vec (Vec Char | Txt)

Note that there are no operators here for overwriting the data pointed at by a persistent value. It is a write-once value, making it much easier to ensure the consistency of the data. Persistent variables are allowed with another type constructor, *Pervar*. However, writing to one of these variables is considered to be a unitary commit operation on the database and is more expensive to perform than the simple persist-operation since it has to guard itself against unexpected failure. In the present implementations, each database is garbage-collected separately off-line to recover inaccessible space. This style of garbage collection means that cross-database pointers are not allowed and any attempt to write a database pointer to a alien database is trapped.

2.4 Infinite Unions

The trusted Ten15 translator will rigorously check all operations to ensure the correct application of the type rules. There are many applications where one would like to do similar type manipulations in other *untrusted* programs and still be sure that the underlying integrity of the type system is maintained. An obvious example is a command interpreter where one wishes to make sure that the correct type of parameters is used with a command which is just some procedure value. It is clearly inappropriate to call up the Ten15 translator to produce the code to obey the procedure. Instead, what happens is that the untrusted command interpreter inherits the type checking facility by using various Ten15 constructions and types; these were, of course, checked by the Ten15 translator when it translated the command interpreter. At the core of this are Ten15 values which are, in effect, the infinite union of all possible types; this type is called for various historical reasons, *Moded*. One can apply a Ten15 operator (to_moded) to a value of any type to get a value of type *Moded*; this is represented by a pair consisting of the original value together with a representation of its type. Typically, a dictionary look-up would require the use of *Moded* values, except in the unlikely case where the possible names all corresponded to values of the same type; a procedure to find the meaning of some name encountered by a command interpreter might have type *(Vec Char → Moded)*. A Ten15 control structure allows one to extract the original value from the *Moded* value, provided that one knows the either the type of this value or one which is greater than it in the lattice of types. Another operation allows one to extract the type as a value of type *Mode* and further operations allow one to explore its structure. In addition, all of the polymorphic Ten15 operations (like assigment or procedure application) are available with *Moded* operands. These operations are effectively interpreted rather than compiled with the type checking done dynamically.

2.5 Polymorphism

The use of infinite unions can be regarded as a kind of polymorphism in which the type of an object is carried dynamically. Any other kind of polymorphism could be implemented in terms of infinite unions; however, efficiency and convenience considerations often dictate the use of more streamlined forms. For example, efficient polymorphism is essential in the definition of the operations of some abstract-data types. The treatment of Ten15 polymorphic types is just an extension of the type changing rules of Ten15, where the representation of the value remaims unchanged. Other examples exist where the same representation could have many different types. For example, a polymorphic identity function, such as one might meet in ML [Gordon 79], would have Ten15 type:

$$\cap_X (X \rightarrow X) \qquad (2)$$

This value could serve as a procedure with type $(X \rightarrow X)$ with any substitution of an actual type for the formal type X; in a sense, the value is lower in the lattice of types than any procedure $(X \rightarrow X)$. Similarly one often wants to have a value that is higher in the lattice than some set of values of similar structure. An example would arise if one wished to keep a vector of procedures of differing types togther suitable parameters:

$$Vec\ \cup_{X,Y}((Ptr\ X \rightarrow Ptr\ Y),\ Ptr\ X) \qquad (3)$$

With some suitable consistent subtitutions, replacing actual types by the formals X and Y, one can construct a procedure-parameter pair and assign it to an element of the vector. It would be nice if this action of substitution in polymorphic types could be included in the general coercion rules so that polymorphic types are indeed least upper bounds and greatest lower bounds as implied by the $\cup$ and $\cap$ notations. Unfortunately this cannot be done without introducing unacceptable restrictions in forming the polymorphic types. Instead Ten15 operations are provided to perform the substitutions explicitly; any implicit type change of a polymorphic value always results in another polymorphic value.

Not all polymorphic values can be created by untrusted programs in Ten15; the rule about not changing the representation in coercions together with efficiency criteria limits one somewhat. These limitations can be summarised, in practice, by saying that the size of the representation of any object accessed must be known at translate time. This means that values of types like (3) can be constructed and used quite naturally. On the other hand, the body of the identity function with type given by (2) cannot be written by an untrusted program since the size of its parameter and result is unknown; however, a related identity function with type:

Π_X *(Ptr X → Ptr X)*

can be written trivially.

The use of polymorphism often sounds rather esoteric; however it has some very mundane applications in the Ten15 PSE. Thus the procedure for linking and loading independently compiled modules (represented by an abstract-data type with constructor *Module*) has type:

Π_X *(Module X → Ptr X)*

This linker was written as an untrusted program and its body contains the use of a data-structure with type similar to (3) to remember which internal modules have already been linked. Its analogue in the Flex PSE is described in [Currie 85a].

2.6 Exceptions

All the operations and constructions of Ten15 are fully defined. Many of them can give error conditions (eg array indexing). Error conditions are usually trapped by means of the trapply operation which is a modification of the normal procedure call operation where:

trapply: *((P → R) * P) → (R | Exception | Void → ⊥)*

The first possibility in the union result of trapply is the result of the procedure call if it ends unexceptionally. The second, of type *Exception* is a value which is characteristic of an error condition and can be queried to recover diagnostic values in the failed call (or any inner ones). The third (a procedure value) will be the result of doing any "long jump" out of the procedure; calling this procedure will complete the long jump.

2.7 Remotes

Using the make_remote operation, one can construct a unique token for an arbitrary value in a particular machine which can then be freely sent round a network where:

make_remote: *X → Remote X*

The token can be used at any time to recover the original value in the machine which constructed it. The most important kind of remote value is the remote procedure. The remote_call operation can be used to apply appropriate parameters to a remote procedure token to call its procedure value in the distant machine delivering its result across the network. In Ten15, this is a very powerful operation since the process of preparing the parameters for transmission will replace any procedure values in the parameters by new remote tokens corresponding to the procedures which could be called remotely by the distant machine. Similarly, procedures in the result value will be new remote procedures in the calling machine. This allows one to construct very general protocols, each one being characterised by the type of the initiating remote procedure. For a more detailed description of this and related topics including the garbage collection of remotes, see [Foster 87].

2.8 Processes

Multi-processing in the Ten15 machine is defined at quite a low level. This is mainly a consequence of our desire to use Ten15 program as a general intermediate language for existing languages; the conflicting requirements of these different languages drives one to use very primitive constructs. Thus, an object of type *Flag* is used as the operand of a simple unitary read-modify-write operation to be used as a primitive in more complex constructions such as semaphores, channels, and the construction of monitors. A *Process* is the result of a launch of a procedure as another pseudo-parallel process; this value can be used as an operand of various operations such a run_process or fail_process so that a user can write his own scheduler for his own processes. The issue of the fairness of allocation of time between users is largely dealt with by the kernel; however the user can allocate his own fraction of time how he pleases.

The rigorous maintenance of type integrity is a considerable problem where parallel processes can have unguarded variables in common; this happens with dismaying regularity in Ada programs, for example. Our current solution to this is to effectively make assignment a unitary operation. The translated code of a Ten15 program only permits a change of process only when the translator knows that it is safe; certainly not in the middle of the assignment of a union value, for example. A more radical solutions will have to be adopted when we attempt to define Ten15 for multiple processors with common memory. For example, by examining the type of the nucleus of a procedure, one could tell whether there are *any* external variables directly accessible to the procedure and hence make it inadmissable to launch as a process in another processor.

2.9 Abstract data types

The representations of values of the above types are effectively defined by the operations that can be applied to them; these operations are all defined as part of Ten15. The only operations applicable to a Ten15 value of some abstract data type (ADT) are all defined by the inventor of the type who chooses a representation for it in terms of existing types. These operations are all expressed in Ten15 (in terms of the representational type) to be applied either as a procedure call or as an open substitution in the Ten15 program. In both cases, the Ten15 translator replaces both types and operations by their representational equivalents before translating so that representational integrity is preserved. For example, linear lists in Ten15 might be abstracted as a type *List X* whose representation is :

```
    L(X) = (Void | Ro_ptr L(X))
with operations:
    cons:(X,List X) → List X
          = λ(x:X, l:L(X)) -> unite_to_L ro_pack(x, l)
    hd: (List X) → X
          = λ(l:L(X)) → Either  l Is field_2 z
                          Then field_1 deptr z
                        Or fail "nil list"
                        End
    ... etc
```

Both of the operators here would have to be implemented as open substitutions of their Ten15 meanings since one cannot construct polymorphic procedures of the correct type (see **2.5**). The inventor of an ADT like *List* can prescribe some coercions between different varieties of *Lists* provided that they are consistent with the permissable coercions on its representation. Thus, with the representation above, the formal *X* behaves in a covariant fashion with *List X* and so a *List [1..10]* could be coerced to a *List [0..99]*. However if the *Ro_ptr* in the definition of *L* were replaced by *Ptr* then no non-trivial coercions would be allowed.

Most of the ADTs in the Ten15 PSE are used to hide details of their representation and limit their operations, rather than in their more classical role of giving representational independence. It is easy to use them in this latter sense if their use is limited to single programs; however, once values of a given ADT are spread across databases and networks the problems of changing its representation become formidable in the extreme. These problems have not been completely solved in Ten15 although tools exist to do transformations in limited cases.

3 Programming in Ten15

3.1 The Ten15 algebra

Various fragments of text have been included to describe Ten15 program and types above. It must be emphasized that these are part of an informal textual notation for Ten15 but are not themselves part of Ten15 programs. Ten15 programs are data-structures, *not* text. These data-structures are abstract data types based on the sorts of an algebra which form part of the formal definition of programs in the abstract machine. This algebra, like any other, is defined in terms of sorts, constructors and laws for the expansion of these constructors. In the Ten15 algebra, the compound constructors give the control structure of the machine and various laws give equivalences between different control structures; for example expressing a for-loop in terms of labels and gotos. There are about 30 significant compound constructors in the algebra with a other less significant ones defining various kinds of grouping. The sorts of the Ten15 algebra include **Type**, **Load**, **Name**, **Operation** etc; To give a flavour of the kind of model intended here, a **Load** occupies the roughly same niche as a statement or expression in a standard programming language and most significant chunks of program would be represented by **Loads**. For example, the equivalent of a declaration in the Ten15 machine as an element of the Ten15 algebra would be a **Load** constructed using:

identity_dec: **Name** * **Load** * **Load** -> **Load** (4)

with the interpretation that the first **Load** is "evaluated"; its value is then used in place of any occurrence of the **Name** in the second **Load**, thus defining the scope of the name. A conditional would be:

cond: **Load** * **Load** * **Load** -> **Load**

where the evaluation of the first load to a *Bool* determine which of the others to evaluate. Another familiar one would be the application of an operation:

operate: **Operation** * **Load*** -> **Load**

ie apply the **Operation** to the evaluation of the **Loads** as parameters. There are approximately 200 of these operations defined ranging from arithmetic to remote access.

3.2 Homomorphisms

One reason for basing Ten15 on an algebra is the discipline that it imposes on any program which itself analyses a Ten15 program; most such analysis programs can be written very conveniently as though they were homomorphisms on the original Ten15 algebra. One example which will arise in any porting of a Ten15 PSE is the Ten15 translator itself. In the current implementations, the Ten15 translator is a homomorphism from the Ten15 program data structure expressed as abstract data types to a function structure whose application gives the translated program for the target in question. Little more than a sketch can be given here of the domains involved. For example, the image of a Load is a function of Ten15 type :

Load = Context → Translation

where *Context* is some type which will contain inherited information like the names in scope and *Translation* will contain the result of translating code for the object machine. The image of a constructor like identity_dec in (4) for example would be:

identity_dec: (Name, Load, Load) → Load

where the first two parameters, together with the parameter of the answer *Load*, will be used to construct a new *Context* with which to evaluate the third parameter.

The use of homomorphisms in the translator illustrate strongly the advantages of this method. The mapping of each construction is independent and is also independent of the order that mappings are applied. This makes maintenance of the translator more tractable and its correctness much easier to determine. It also means that a translator for one target host machine serves as a very good example for writing one for a new host.

The Ten15 translator is an extreme example of the use of a homomorphism to deduce properties of a Ten15 program; however, the same simple homomorphic framework can be used to evaluate other properties. For example a trivial homomorphism allows one to evaluate, say, the set of names declared but not used; a less trivial one could give a function which is a pretty-printer for the program. Also of considerable interest are those homomorphisms which are transformations of Ten15, eg those which apply some of the laws of the algebra such as replacing all for-loops by their expansions in terms of labels and gotos. We see this kind of program transformation as the first step in more general program proving; however this requires other tools like theorem provers which we do not possess as yet.

3.3 Properties of Ten15 programs

Ten15 programs are intended to be translated into the code of some host computer and run in a Ten15 run-time system. This run-time system could be implemented on a bare host; more usually it will be embedded in some existing operating system on the host. In the latter case the security and integrity of the Ten15 system is only as good as that of the host system; however the practicalities of producing things like device drivers make it a virtual necessity.

The integrity of the Ten15 run-time system (and hence the security of anything built upon it) depends on the typing rules never being violated. In other words, a value in store must have the structure implied by its type; for example a *Ptr* type must always point to a real block in mainstore. Any mis-alignment of this structure could result in mayhem; anything from the garbage collector going into an endless loop to a database being corrupted. This has considerable effect on the kind of control structures and operations in Ten15. At a trivial level, one can see that a variable *must* be initialised to a value of the required type, so that any fact that the translator might deduce from its type is correct. In fact, all declarations in Ten15 follow the pattern shown by identity_dec in (4). The type of most constructions (a notable exception being procedure bodies) is not explicit but is deduced by the translator; for example the type of a conditional expression is the least upper bound of the types of its arms. Every operation is defined completely either by delivering a value of some type depending only on the types of its operands or by causing some defined exception; there are no "undefineds" in Ten15. Indeed, this applies to all constructions since the underlying bias of Ten15 is towards expression evaluation rather than state changes.

As well as being a programming mechanism in its own right, Ten15 is also the target for all compilers for standard languages used in the system. This means that Ten15 must be able to cope with all constructions likely to be found in those languages, not just some common subset eg exception and process handling operations are defined. Similarly, labels, branches and jumps all form part of various Ten15 constructions with rules for scoping rather like those for value and variable names. The use of labels is, in fact, essential to implement efficiently the many slightly different variations in similar control structures in different languages. The efficiency of programs written in standard languages compiled via the Ten15 route is an important factor. We hope that it is comparable to those produced by compilers native to the host machine; however we have not yet done any extensive rigorous comparisons.

3.4 Implementation

In the current implementations, mainstore is allocated linearly in blocks with a garbage collector applied when some limit is reached to compact live blocks. Most of these blocks will correspond to the construction of some Ten15 value. A pointer is such a value; a value of type *Ptr X* is implemented as an address of a block containing a value of type *X*. To be able to carry out a garbage collection, words in the host machine which contain these block addresses are distinguishable from those containing non-addresses by using a bit or two of the words as tags. Thus, on the Vax implementation, the least two significant bits in each 32-bit word are used for various tag purposes. This has some consequences on the representation of scalar data; for example, integer ranges representable in one word are limited to 30 bits. The type rules of Ten15 will ensure that any translated program will never confuse addresses and scalars or misuse these tag bits in any way. The time taken for one garbage collection depends, of course, on the mix of data present. However, it operates in a time which is linear with respect to each of variables of this mix, eg, the number of live blocks or the total area of store available. With a typical mix on a single-user Vax-station, it takes just over a second for a area of 2 megabytes.

As mentioned before, the Ten15 translator is itself expressed in Ten15. It relies on the existence of a run-time system to run the code that it produces. This runtime system, often called the kernel, is in three more or less distinct parts. The first part is the set of run-time routines required by Ten15 to implement things like the mainstore storage allocation and garbage collector. Code produced by Ten15 translator will access routines in this part directly at the level of the host computer's instruction set and all of this part is written using some tools of the host eg the code assembler of the host. The second part is entirely written in Ten15 with a few privileged operations to cheat the type rules. This part includes the routines which handle the run time representations of types themselves; also in this part are the various "flattening" procedures for preparing data structures to be kept in a database as a *Persistent* object or sent across a network as a remote object. The third part is only accessed by other routines in kernel and consists of device drivers; usually they will be implemented by making primitive calls on some host operating system. On current implementations, the total size of the kernel is less than 100 kilobytes and the amount not written in Ten15 just a few kilobytes. The size of the mainstore garbage collector, for example, on Vax is about 200 MACRO instructions.

Porting a Ten15 system to a new host will require work to create a new Ten15 translator together with that part of the kernel not expressed in Ten15; the rest consists of re-translating the Ten15 part of the kernel and the various programs and tools of Ten15 PSE. These latter programs and tools have been written in a variety of languages and translated to Ten15.

4. Conclusion

Two different implementations of Ten15 are in progress at present. One is a prototype running on the Flex system and the other is on Vax running VMS. Compilers, using the Ten15 route, for Ada™ [Ichbiah 83], Algol68 RS [Woodward 82] , Pascal[BSI 82] and a notation for Ten15 are more or less complete while others including one for ML [Gordon 79] are projected.

Many of the properties of any PSE running in a Ten15 machine can be inferred from the broad overview given above and some of the more important ones have been highlighted in the text. The current implementations of Ten15 PSEs are largely based on the Flex PSE; the type structure of Ten15 taking the place of the capability structure of Flex. Some aspects of the Flex PSE are detailed in [Currie 85a]. Probably the most important feature of both is the way that one can use values and program in ways that were unanticipated when they were created. This is largely a consequence of having first class procedure values and a common addressing space across all programs; the integrity and security of those facilities being enforced by the type structure in Ten15 and the capability structure in Flex.

References

[BSI 82] "Specification for the computer programming language, Pascal" BS 6192:1982 British Standards Institution

[Currie 81] I.F.Currie, P.W.Edwards and J.M.Foster: Flex firmware, RSRE Report No. 81009, 1981.

[Currie 82] I.F.Currie and J.M.Foster: Curt: the command language for Flex RSRE memorandum No. 3522, 1982

[Currie 85] I.F.Currie, P.W.Edwards and J.M.Foster: PerqFlex firmware RSRE Report No. 85015, 1985.

[Currie 85a] I.F.Currie: Some IPSE aspects of the Flex project "Integrated project environments" J. McDermid (Ed). Peter Peregrinus 1985

[Foster 82] J.M.Foster, I.F.Currie and P.W.Edwards: Flex: a working computer base on procedure values Proc. of international workshop on high-level architecture, pp 181 185, Fort Lauderdale, Florida (Dec 1982)

[Foster 87] J.M.Foster, I.F.Currie: Remote Capabilities to appear Computer Journal 1987

[Gordon 79] M.J.Gordon, A.J.Milner, C.P.Wadsworth "Edinburgh LCF" Springer-Verlag (Berlin 1979)

[Ichbiah83] J. Ichbiah et al. "Reference Manual for the Ada programming Language" U.S Dept Of Defence 1983. Ada is a registered trade mark of the US DOD

[Woodward 82] Edward Arnold(London 82) P.M.Woodward,S.G.Bond,"Guide to ALGOL 68"