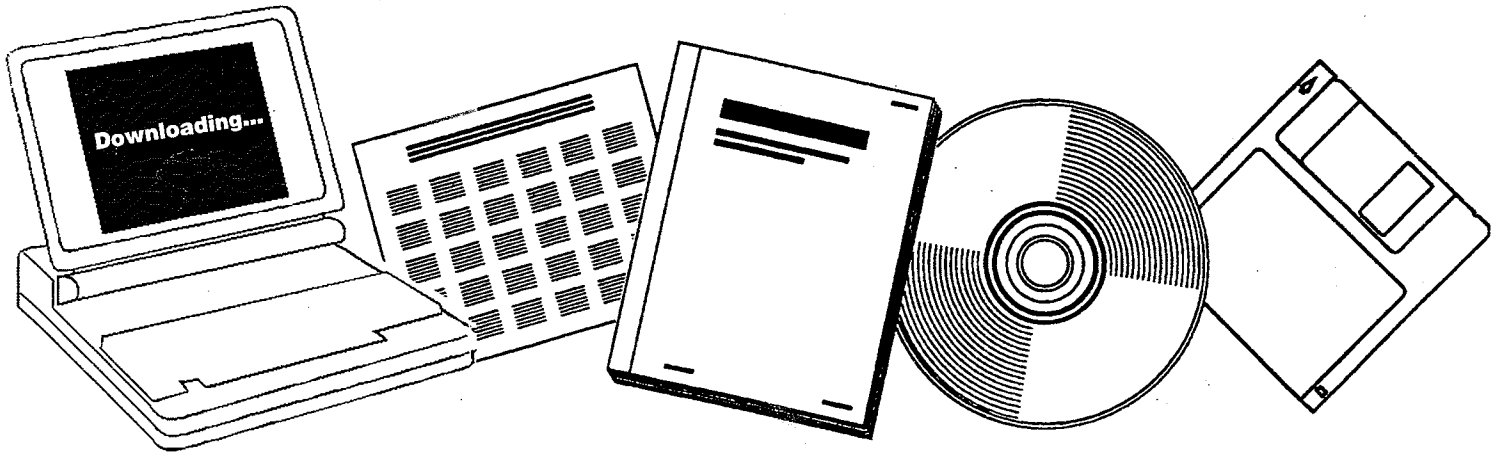
TEN15: AN OVERVIEW

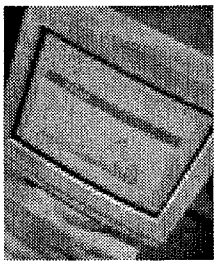
ROYAL SIGNALS AND RADAR ESTABLISHMENT,
MALVERN (ENGLAND)

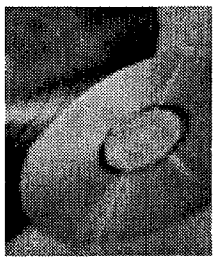
SEP 1986



One Source. One Search. One Solution.

NTIS





Providing Permanent, Easy Access to U.S. Government Information

The National Technical Information Service is the Nation's largest repository and disseminator of government-initiated scientific, technical, engineering, and related business information. The NTIS collection includes almost 3 million information products in a variety of formats: electronic download, online access, DVD, CD-ROM, magnetic tape, diskette, multimedia, microfiche and paper.

Search the NTIS Database from 1990 forward

More than 600,000 government research information products have been added to the NTIS collection since 1990. All bibliographic entries for those products are searchable on the NTIS Web site at www.ntis.gov.

Download Publications (1997 - Present)

NTIS provides the full text of many reports received since 1997 as downloadable PDF files. When an agency stops maintaining a report on its Web site, NTIS still offers a downloadable version. There is a fee for each download of most publications.

For more information visit our website:

www.ntis.gov

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Technical Information Service
Springfield, VA 22161

RSRE
MEMORANDUM No. 3977

ROYAL SIGNALS & RADAR ESTABLISHMENT

TEN 15: AN OVERVIEW

Authors: P W Core and J M Foster

PROCUREMENT EXECUTIVE,
MINISTRY OF DEFENCE,
RSRE MALVERN,
WORCS.

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Memorandum 3977

Title: **TEN15: AN OVERVIEW**

Authors: P.W.Core and J.M.Foster

Date: September 1986

Summary

Ten15 is a strongly typed, algebraic abstract machine. It is in use as the target machine of several compilers. The method of definition makes it possible to use formal techniques of algebra as a basis for tools. This memorandum contains an informal description of the machine and its programs.

CONTENTS

1. INTRODUCTION

Ten15 is an abstract machine which is defined algebraically. It is strongly typed, using a universal system of types.

- Ten15 is the target machine for a number of compilers. At present compilers for Pascal, Ada, Algol68, ML and a direct notation for Ten15 are in various stages.

- Programs for the Ten15 machine can be translated into machine code. A translator for the Flex computer[1,2] has been in use since 1984 and a VAX implementation is being started.

- Since Ten15 is defined algebraically, algebraic techniques are available to construct tools to operate on programs. The VAX translator is being written as a homomorphism. Analytic programs to determine properties of pieces of Ten15 can be algebraically implemented.

- Algebraic techniques can also be used synthetically to construct Ten15 programs. This is our intended approach to "program re-use".

- The system of types used by Ten15 is universal. The conventional types in programming languages restrict their type checking to one statically linked program. Ten15 types permit the checking of the dynamic linking of unanticipated procedures and embrace the operating system and the translator itself.

- The type checking in Ten15 is more stringent than the capabilities of the Flex machine, yet it is equally flexible.

- The use of Ten15 as a common target for compilers means that programs from different sources can cooperate and a single system of tools can be applied to them. The code produced is acceptably efficient. Using the Ada compiler through Ten15 to Flex yields code which is approximately equal in space and speed to code produced by an ad hoc compiler for Algol68 on an equivalent program.

- The Ten15 types are sufficient to describe backing store and network values, and appropriate operations are available upon these.

The Ten15 machine has programs, which can be obeyed in a state of the machine, producing another state. Ten15 programs are defined as members of a particular algebra, the program algebra. The states and transformations of a Ten15 machine are also defined algebraically. We have chosen to make the effect of obeying a Ten15 program legal for every state (including inappropriate states and non-terminating programs), so a program is associated with a total function from states

to states. The formulation of universal algebra which we use is derived from that of the ADJ group[3].

In order to define the Ten15 machine we must describe the program algebra and the association between programs and state-to-state functions. Formally this association is defined as a homomorphism, but in this paper we are giving an informal account of Ten15. All the constructions will be treated in enough detail to give an initial understanding, but not enough to create an implementation.

In writing this description, we must especially acknowledge Ian Currie, one of the designers of Ten15, and the RSRE staff who have worked on Ten15.

2. TEN15 MACHINE STATES: VALUES AND MEMORY

The basic constituents of a Ten15 state are values and a memory. A memory is a repository for values. The whole memory is changed by an assignment. References into the memory are themselves possible values.

Ten15 values are typed (moded, following Algol68 terminology), every value having an associated type which determines the operators which can be applied to it. A type corresponds to a set of values, but these sets are not all disjoint. Many of the types are close to those required by high level languages so that the Ten15 values representing values in high level languages are usually obvious.

The values are divided into those which do not involve memory (the scalar values), those which are pure memory values, those which are mixed, values in backing store and network values (which are not described here), and those values which are involved in procedures and exceptions, the description of which is deferred to the next section. In addition Ten15 permits the introduction of new user-defined abstract types. The values belonging to new abstract types are represented by values belonging to previously defined types.

The different types of value are described below. In this paper the names and constructors for types are underlined.

A few of the operations on the values are described here, but the full list of operations is given in Appendix D.

2.1 Scalar values

void

void is a primitive mode. There is only one value of mode void.

bool

bool is a primitive mode with two values. There are the usual operations on boolean values,

range

Ten15 does not have a mode integer, but instead the mode constructor range. A range mode has two integer parameters, the first less than or equal to the second, and every value belonging to the range lies between these as inclusive limits. For example, a representation of characters might be range(0, 255).

In all operations delivering ranges,such as '+', '-', etc., the range of the result is the smallest range including all possible results, given the ranges of the operands. For example a value of mode range(1,10), when added to a value of mode range(1,100) delivers a value of mode range(2,110).

real

real is a mode constructor, designed to represent usual implementations of real numbers. A real modes has two integer parameters; the first represents the number of bits in the mantissa, and the second the number of bits in the exponent. Exponents are assumed to be base 2. The operations on reals are conventional and the mode of the result is the real mode constructed from the maximum of all mantissa sizes and the maximum of all exponent sizes.

This form of real modes allows reals to be transferred to another machine, be operated on and returned to the original machine in the same representation.

2.2 Memory Values

Memory in Ten15 is entirely controlled by Ten15 values. If there are no Ten15 values with access to memory then there is no ability to access the memory. The memory is organised on a capability basis, that is, memory can only be accessed if a capability, a value with access ability, for that part of memory is held. Capabilities can not be forged, so by controlling the distribution of a capability the memory which it has access to is also controlled.

pointer

pointer is a mode constructor, and the modes generated are constructed from one mode parameter, for example pointer(bool). The values whose mode is constructed using pointer are values with access to memory, and the mode from which they are constructed is the mode of the value in memory to which they have access.

Pointers are generated by the operation 'generate pointer' which operates on any value, generates a memory location in which to place it and then generates a value, the pointer, to access it. This pointer gives the ability to replace the value to which it points with any other

value of the same mode. It also gives the ability to read the value to which it points.

Capabilities not generated from sequences of operations on this pointer will not have access to the memory location. Thus by limiting the access of the pointer, control of access over the memory location is maintained. Pointers are mainstore values which are lost when the mainstore is lost.

reference

reference is a mode constructor, and the modes generated are constructed from one mode parameter; reference(pointer(bool)) is an example. The values whose mode is constructed from reference are values with access to memory, and the mode from which they are constructed is the mode of the value in memory to which they have access. They are very like pointers.

References can be created in many ways, but always from other memory values. Hence they inherit their memory capabilities from some other object which also has memory capabilities. The simplest manner of creating a reference is from a pointer. A value of mode reference m, a reference, can be created from a pointer of mode pointer m. Like a pointer a reference gives the ability to replace the value it references with another value of the same mode, and also to read the value referenced. But whereas a pointer can only be created by generation, and always refers to the whole of the object in question, a reference can access parts of an object. For example a

reference(struct(bool,range(0,255)))

could be selected from to give either a ref(bool)) or a ref(range(0,255)) each of which can be altered or read. They would share store with the original reference, so if an assignment is made to the ref(bool) and the original was read, the boolean field of the structure would have the new value.

vector

vector is a mode constructor, and the modes generated are constructed from one mode parameter and a pair of integers, for example vector(range(0,255),0,10) may be a vector of characters. Vectors are memory capabilities with access to a number of values of the same mode, the mode from which the vector was constructed. The number of components in the vector lies between the two integers in the mode construction (inclusively).

Vectors are created in two ways. Firstly, a vector(m, l, u) can be made from a value of type m to initialise the vector and a value of range type to specify the number of elements. The vector then contains

a number of copies of the value. Secondly a structure with n fields, every field having the same mode, m, can be used to produce a vector(m, n, n) containing the elements of that structure.

Vectors can be reduced in size by trimming, a trimmed vector references a subset of the original vector. The trimmed result shares store with the original. The components of one vector can be assigned to another vector of the same mode, provided that they are the same size.

There is an operation on a vector giving the number of elements in the vector. For example, when applied to vector(range (0,255),0,10) a value of mode range(0,10) is delivered.

Vectors can be indexed by a range value to give a reference to an individual component. If the index is outside the range of the vector, then an exception is delivered. The reference provides a means of altering and reading individual values independently of the vector. The memory locations referenced can also be changed by assignments to the original vector as a whole, or to trims of the original vector.

array

array is a mode constructor, modes generated from it are constructed from an integer, which is the dimension of the array, a mode (the mode of the elements referenced by the array) and a pair of integers for each bound in the array.

Arrays are similar to vectors, but they have a different indexing mechanism. Each dimension has a lower and upper bound, lying in specific ranges determined by the integers in the mode construction.They are indexed to give references, but one index is required for each dimension to deliver the reference. If the index of each dimension does not lie within the bounds of that dimension then an exception is delivered.

Each dimension of an array can be trimmed individually in a similar manner to vectors and an array can be sliced (have its dimension reduced by one by indexing in only one of the dimensions). There are array operations, similar to that operation on vectors which gives their size, to give the upper and lower bounds of a specified dimension.

read only memory values

For all the memory access values just described, there are 'read only' versions. These can be read, but cannot be used to alter the memory.

They are generated from any memory value. They are not to be assumed to be constant because they access the same memory as the value from which they were created. This value may have the ability to change the memory.

2.3 Mixed Values

struct

The Cartesian product of any number of modes is a mode, which is written as struct(m1, m2, ... mn). The values of this mode are n-tuples of values of the constituent modes. For example the Cartesian product of a bool and a range(0,255) is a struct(bool, range(0, 255)). The elements of a structure can be selected from the whole.

union

The disjoint sum of any number of modes is a mode, which is written as union(m1, m2, ... mn). The values of this mode can be represented by a tag to determine to which of the modes the incumbent belongs, together with the value itself. There is an operation which takes a tag and a value, uniting the value to the mode indicated by the tag. This will not be allowed if the mode indicated is different from that of the value. For example the value true can be united to either the first or third fields of the union union(bool,range(0,255),bool), but not the second.

This type of union is different from the Algol68 union, which is an approximation to set union in which it is not possible to have a mode repeated in the union. It is also different from the Pascal variant record which corresponds to reference to union and which allows the assignment of the tag independently of the value.

choice

choice is a mode constructor, and the modes generated are constructed from one mode parameter, for example choice(pointer(bool)).

There are operations to convert any mode, m, to a choice(m), and there is an operation to convert void to any choice(m) . To obtain the value in a choice there is a test at execution time to determine whether the value in a choice is void or not, and to obtain the value if it is not void.

Choices are included in Ten15 so as to provide a convenient representation for references in languages such as Algol68 where a 'NIL' referenece is required; a REF X being represented as say a choice(reference(X)) with NIL being the void option.

moded values

There is another primitive mode, moded, which can be thought of as the value bound to its mode, or alternatively as the infinite union of all modes. Very few operators can operate directly on moded values because the mode of the value is not known until execution time. There are two ways of obtaining the value bound into a moded. Firstly there is

an execution time test to assert whether or not the moded value is of a specific mode, if so it delivers the value, otherwise an exception is delivered. Secondly there is the case_moded construction described in the following sections. Each of these is related to the corresponding operation on unions.

2.4 Filestore Memory Values

In addition to mainstore memory, it is necessary for Ten15 to encompass the notion of filestore memory. This is in order that Ten15 can be used in the description of an entire system. It is not assumed that there is only one filestore. The values for accessing filestore values are filestore capabilities, with similar properties to mainstore capabilities. There are two types of filestore capability.

persistent

The mode of a persistent value is generated using the constructor persistent and any mode.

Persistent values are generated in a similar manner to pointers with a 'make persistent' operator, operating on any value, generating space for a filed representation of it on filestore, and generating a value to access it. It is not possible to assign to persistent values, only to create them and read them, so that they can be considered constant or write-once values.

When reading a persistent value, a copy of the value stored is delivered. For static values this is indistinguishable from the original. But for memory values a copy is not equivalent to the original, in that assignment to it does not change the original, nor vice versa. The copy will be internally consistent, in the sense that if the same mainstore capability is used more than once in the value which is output to filestore, then the value read from filestore will maintain equality between those capabilities, but not with any others which may still exist in the mainstore memory.

Though a persistent version is available for values of any mode, it is not possible to put on one filestore values belonging to another.

persistent_variable

The mode of a persistent_variable value is generated using the constructor persistent_variable and any mode.

Persistent_variable values are generated in the same manner as persistent values. They can be assigned to and dereferenced. they contain only persistent values from the same filestore.

2.5 Abstract Data Types and New Constructors for Types

As well as the mode constructors just listed, it is possible to make new

types and new type constructors which can be defined by the user. Each new type or constructor must have a representing mode and a set of atomic operations which are expressed in terms of the representing mode. For example it may be desirable to have a new constructor list so that Ten15 can handle, for example, values of mode list(bool) or list(range(0,255)) and list(x) could be represented as

$$r \text{ where } r = \text{choice}(\text{ptr}(\text{struct}(x, r)))$$

and x is a formal mode. The appropriate operations might be head, tail, cons and is_empty.

2.6 Network values

These are not described in this paper.

3. *EVAL*, PROCEDURES AND EXCEPTIONS

3.1 Procedures

In order to evaluate a piece of Ten15 it is necessary to know the values which correspond to the names in it which are not locally declared. Given this environment, we have a function from states to states. So the Ten15 *eval* function is a mapping from the Ten15 program algebra to functions from environments to state transformations.

$$(\text{NAME} \longrightarrow \text{VALUE}) \longrightarrow (\text{MEMORY} \longrightarrow \text{MEMORY} \times \text{VALUE})$$

Here (NAME → VALUE) gives the name-value associations of the environment. The (MEMORY → MEMORY × VALUE) is the effect of execution.

An illustration from Algol68 is the expression

$$x := 3$$

This can be expressed in the above form, where the external value (a reference to an integer) associated with the name *x* needs to be supplied. Execution produces a change in the memory accessible by the external value, leaves the rest of memory alone, and delivers the value of x.

The ability to treat programs as data is inherited by the Ten15 program itself and is one of the more important features of Ten15. It does this by means of values of type protocode together with values of type package and procedure. In the run_time system a protocode corresponds to a piece of Ten15 program and so it needs to be bound to values (its version of an environment) before it can be run. When it is so bound it yields a procedure. When this is run, it will produce a

changed memory and a value, the result of the procedure. It will be shown that Ten15 programs do not contain values, and particularly not values which address the memory (literal constants are treated differently). This means that Ten15 protocode values are portable to other Ten15 machines, as all Ten15 machines are driven by the common Ten15 algebra. Properties of protocode can be analysed independently of the Ten15 machine states, in particular independently of the memory.

A protocode, then, needs to access externally supplied values. Indeed if it does not do so, then it is an entirely self-contained object, without parameters or access to anything non-local. Such programs are possible, but very rare.

There appear to be three times from the creation of a protocode until its execution at which it would seem appropriate to supply values. The first is at the creation of the protocode. For example a piece of program may require the use of previous defined modules. It is generally known when the protocode is created that it is only intended to be run with specific modules and no others. We call these the constants of the protocode, though the name is slightly misleading. The result of supplying these values gives a package.The second is during loading or execution of some other piece of program, when certain non-local values may be bound in. The result of binding the non-locals into a package gives a value of type procedure. The third time is just before execution of the program itself, when the parameters are supplied.

protocode + constants ⟶ package
package + non-locals ⟶ procedure
procedure + parameters ⟶ executable program

Applying a Ten15 procedure to its parameters immediately causes execution, and delivers the value which this produces. protocode are therefore constructions in the Ten15 algebra which require precisely three external values, the constants, non-locals and parameters and in consequence are supplied with three names.

For example consider the piece of Algol68

```
REF INT count = HEAP INT := 0;

PROC add = (INT i)INT: count +:= i;

add(3)
```

The procedure *add* could be represented by a Ten15 procedure. It is dependent on only two values, *count* and *i*. *i* is the name of the parameter, with which it is supplied immediately before execution. The appropriate value for *count* does not exist until the algol68 program is

executed, in particular it is not known at compile time. However it does exist when the procedure is created as the second line is obeyed. So *count* is the name of the non-local. There are no constants. The Algol68 program would generate a reference *count* as the first line is executed. This is then bound to an anonymous package, to give a procedure representing *add*, which then finally is supplied with the value *3* as a parameter before execution.

If this procedure were part of another procedure for example,

```
PROC add3 = (INT i)INT:
BEGIN
  REF INT count = HEAP INT := 0;

  PROC add = (INT i)INT: count +:= i;
  add(i);
  add(3)
END
```

then the value of the protocode from which *add* is constructed is known when compiling *add3* and can thus be supplied as a constant to *add3* at compile time.

In general the protocode, in terms of the Ten15 algebra, will involve applying operators to values. The types of these values may be known from the examination of the Ten15 structure and in such cases it is known whether or not the value is of the correct type in order for the operator to be applied to it. In the Algol68 example it is known that *count* is a REF INT and *i* is an INT so that *+:=* is a valid operator to apply to these operands. If the operator could not be applied to the value then at execution an exception would have to be delivered. Since it is known whether or not exceptions of this nature will occur at execution by examination of the Ten15 algebraic structure, the same effect could have been obtained by an explicit failure. If the specification of a program involves the application of operators to values of the wrong mode, then this is normally an error, which we treat differently. So protocode can only represent Ten15 structures which guarantee not to produce this exception on execution. This is similar to saying that the compiled code of an Algol68 program will not produce any modal errors.

The constants, non-locals and parameters are also available to the protocode at execution and it is necessary to know the mode of these values in order that the application of operators can be guaranteed not to produce an error; in the *add*, if the modes of *count* and *i* were not known, it would not be possible to determine whether or not the application of *+:=* was valid. The mode of the protocode is thus constructed from four values; the modes of the constants, the non-locals, the parameters and the result. When a protocode is supplied with its constants, their modes are checked to ensure that they correspond with the mode specification of the protocode and a

package is delivered whose mode is constructed from three modes, the non-locals, the parameters and result. Similarly the procedure is constructed from only two modes after the checking of the mode of the non-locals against the mode specification of the package.

add was constructed from a protocode of mode

protocode(void, ref int, int, int)

void was supplied at compilation to give a value of mode package(ref int, int, int), which was supplied with the non-local *count* to give a proc(int, int) which was finally applied to *3* to deliver a value of mode int. The mode of the protocode from which *add3* is constructed is

protocode(package(ref int, int, int), void, int, int)

One of the most important kinds of value which can be bound in this way is a value of type protocode itself. Consider an Algol68 procedure

```
PROC f = REAL:
BEGIN REAL r = random;
    PROC g = (REAL t)REAL:
    BEGIN r + t END;

    g(g(pi))
END
```

The internal procedure g has to be created from a package which needs *r* binding to it when *f* is obeyed. That package in turn was created from a protocode, *p*, by binding void to it. So *f*, which had to be created from an original protocode, needed to have *p* bound to it as a constant. This is why the Ten15 signature contains no operations for creating protocode values. They are created and bound in by an external mechanism.

This notion of protocode adds one more binding time for values to that of Landin's closures[4]. This is not a fundamental change, since the new notion could have been expressed in terms of the old one, and it is not clear why we should stop at two binding times. The present decision is just a pragmatic one.

3.2 Polymorphic Values

Ten15 contains some generic operators, by which is meant, operators that are applicable to values of differing mode. One example of such an operator is trim_vec, which trims a vector. This is clearly applicable to a vec bool, vec range, in fact to any vector. It is possible to specify Ten15 protocodes which will guarantee not to have any modal

errors, but also be applicable to constants, non-locals and parameters of differing modes. For example, a protocode which took a vector as a parameter and delivered a trim of it as a result. In order not to restrict the mode specification of the protocode so that a new protocode is required for every mode of vector which may be required to be trimmed, Ten15 has the notion of generic protocode. The mode specification of the protocode requires the form of the parameters, and the protocode will again guarantee that no operator is applied to a value of the wrong mode form. The mode of such a generic protocode may be

poly(x, protocode(void, void, vec x, vec x)

where poly means polymorphic, and x is a formal mode to specify the form of the protocode.

3.3 Exceptions and labels

Certain constructions in Ten15, the assertions, allow of two possible outcomes. Either, like operators, they modify the state and continue, or else they detect some condition. In the latter case they can be thought of as having the effect of conditional jumps, though the algebraic view uses different terminology. It thinks of labelled statements as being equations. So the Algol68

```
lab: x := x + 1; IF x < 30 THEN GOTO lab FI
```

is thought of as the equation

```
lab = x := x + 1; IF x < 30 THEN GOTO lab FI
```

that is, *lab* is defined to mean that particular state transformation, which in turn involves *lab*. To give a meaning to this we have to solve the equation. In terms of execution, this comes down to the normal meaning. In Ten15, labels are introduced in a very constrained way, which makes it clear what equations have to be solved. This rules out the direct use of this construction for jumping out of procedures, though an algebraic expression of this idea is included using different means.

An alternative form of assertion allows the detected condition to generate, not a jump to a label, but an exception. Once again an algebraic understanding of this is needed, and is provided by the use of exception values (of mode trap). The exceptions produced by the erroneous use of instructions which can only be detected at execution time, are handled in the same way. This covers such errors as overflow and index out of bounds.

4. CONSTRUCTIONS IN THE TEN15 ALGEBRA AND THEIR TRANSLATION

The Ten15 algebra is a many sorted algebra and the translator is a homomorphism from constructions in that algebra. To specify the theory of the algebra we have to give the sorts involved and the arities of the operations. These are tabulated in Appendix A. We will introduce the more important operations gradually in this section, as well as the more important sorts. Sorts are written in capital letters, and the most important is LOAD, which is the sort roughly corresponding to the notion of statement or expression. Ways of constructing LOADs are the main concern of this section. The full list of LOADs is given in Appendix B.

The translation of structures in the algebra of sort LOAD delivers functions in the Ten15 machine of the form

$$(\text{NAME} \longrightarrow \text{VALUE}) \longrightarrow (\text{MEMORY} \longrightarrow \text{MEMORY} \times \text{VALUE})$$

The function from NAME to VALUE will be called the environment. The 'execution' of a load will mean the effect of executing a translated load, the environment of execution being explicitly stated; and all changes in the memory from executing a load are compounded, that is if one load is executed before another, the memory delivered from that load is passed as a parameter to the other.

load_void

This is a primitive load. Its translation is independent of any external values and hence of any name-value associations. It does not alter the memory of the machine and on execution delivers a void. The translation of *load_void* is

$$\lambda\text{env}.\ \lambda\text{m}.\ (\text{m},\ \underline{\text{void\_value}})$$

load_bool(BOOL *b*)

This load in constructed from a boolean constant. Its translation is independent of any name-value associations, does not alter the memory of the machine and on execution delivers *b* with mode bool.

load_name(NAME *n*)

This load is constructed from a name. Its translation does not alter the memory, but is dependent on the environment. If *n* is in the environment, then the value with which it is associated is delivered; otherwise an exception is delivered. Since, in the construction of a protocode, the names in the environment are known, it is possible to determine from the Ten15 structure, whether this type of error would occur. For this reason only structures in the Ten15 algebra, together

with sets of names, which do not produce this error are allowed in formation of protocode.

identity_declaration(INTRODUCTION *n*, LOAD *def*, LOAD *scope*)

An INTRODUCTION consists of a name and a string. The string gives the characters which are to be used to represent the name in printed form. The intention is to use the name for quick comparisons, and to use the string in pretty printing or structure editing. The execution of *identity_declaration* in the context of its environment delivers the value produced by first executing *def* in the same environment, associating the value delivered with the name part of *n*, and then executing *scope* in the context of the original environment with this new association added, using the changed memory. The total change to the memory is that produced by the change in the memory from executing *def* followed by the change from executing *scope*.

Parallel Algol68 example:

```
(INT x = y+3; x+y)
```

x would be the INTRODUCTION *n*, *y+3* would be the load *def* and *x+y* would be the load *scope*. *y+3* is evaluated in the context of some environment including *y*. The INTRODUCTION *x* is then associated with this value, and the value associated with it can be accessed via this name in *x+y*, as well as any other values in original environment.

A further example, involving assignment is

```
(INT x = (t := 3; y + 3); t + x)
```

in which the load *t+x*, the *scope*, is obeyed using the memory as changed by *(t := 3; y + 3)*.

load_sequence(SEQLOADS *seq*)

The execution of *load_sequence* in the context of its environment causes the execution of all the loads in *seq* in turn, the change to the memory is the accumulation of the memory changes in each of the loads, and the value delivered is the value delivered by the final load. Except for the last load all the values delivered by the loads must deliver void; an exception is delivered if any of these values is not void.

Parallel Algol68 example:

```
y := 4; x := IF y > c THEN y ELSE c FI; y
```

The value delivered by this series is *y*, but the memory changes of all the expressions take place. Algol68 is different from Ten15 in that it does not require that the intermediate expressions in a series should deliver void.

load_tuple(TUPLELOADS *tuple*)

The execution of *load_tuple* is almost identical to *load_sequence* except that instead of disregarding all but the last of the delivered values, it delivers an n-tuple of them which has the mode struct(ma, mb, .. mn) where ma, mb, .. mn are the modes of the values delivered by the execution of the loads in *tuple*.

Parallel Algol68 example:

```
(x=y,a > b,IF a > b THEN a ELSE b FI)
```

would deliver a struct(bool, bool, range(l, u)), if *a* and *b* are names for values of mode range(l, u).

case_integer(LOAD *control*, CASE_LIMBS *limbs*, LOAD *out*)

The *case_integer* construction is designed to represent the selection of an expression to be evaluated, dependent on the value of a controlling integer. The execution of *case_integer* in its environment causes the execution of *control* in the same environment. The value delivered must be a range, otherwise an exception is delivered. Each CASE_LIMB is constructed from range modes and a LOAD. The CASE_LIMB containing the range in which *control* lies is determined and its load is then executed in the same environment as *control*. The value delivered from this load is the value delivered by *case_integer*. If the value delivered by *control* does not lie in any of the ranges in *limbs* then *out* it is executed in the same environment as *control*.

case_union(LOAD *union*, UNION_LIMBS *limbs*, LOAD *out*)

The *case_union* construction is designed to decompose a union type value. The union type value in Ten15 is a disjoint sum, essentially represented by a tag to determine in which field of the union the value lies and the value itself. The *case_union* construction executes different loads depending on the tag and makes the value in the union available during the execution of that load. *union* is executed and if the type delivered is not a union, then an exception is delivered. Suppose that the value has tag, *t*, and that *v* is the value from which the union value was made. Each UNION_LIMB is constructed from an INTEGER, an INTRODUCTION and a LOAD. The UNION_LIMB owning the integer equal to *t* is determined and its load is then executed in a new environment. This consists of the original environment of *case_union* plus the name-value pair which holds the name of the limb and *v*. If the tag of value delivered by *union* does not match any of the integers in *limbs* then *out* is executed in the same environment as *union*.

case_moded(MODEDS *modeds*, MODED_LIMBS *limbs*, LOAD *out*)

The *case_moded* construction is the means in Ten15 for handling modes at execution time. For example in an operating system it may be necessary to have a construction which allowed the application of a procedure to some parameters, provided that they were of the correct mode. The ability to do this does not exist in languages such as Algol68 where procedure application is only allowed when the mode of its parameter and result is known. Nor is it available in ML. The *case_moded* construction gives the ability to assert relations between the modes of values whose modes are not known until execution time. To apply a procedure to its parameters, it is only necessary that the mode of the parameter and the parameter mode of the procedure are the same. This can be expressed by saying that mode of the pair, (function, value) has the form $\lambda x,y.\ (proc(x,y),x)$. The asserted mode relationship between two values holds if there exists a substitution of the formal modes giving the modes of the values.

The *modeds* are executed in order in the context of the environment of *case_moded*. If any one of them is not a moded then an exception is delivered. A MODED_LIMB consists a mode relationship, a vector of INTRODUCTION, and a LOAD. The set of modeds are compared with the mode relationship in each limb in turn until one is found where there is a substitution of formal modes to match the actual modes in the modeds. The values in *modeds* are then made available during the execution of that limb by associating them with the names of that limb before execution. If the modes of the values delivered by *modeds* do not match any of the mode relationships in *limbs* then *out* is executed in the same environment as *modeds*.

An example is

```
case_moded proc, parameter in
  (formal x: f|proc(x,bool), p|x) f(p)
out
  fail( "mis - match of parameter or procedure" )
end_case_moded
```

proc and *parameter* are modeds. If *proc* is a procedure delivering a bool and *parameter* has the mode of the parameters of that procedure then the limb is executed, applying the procedure to the parameter,with the procedure accessed via the name *f* and the parameters accessed via the name *p*. If this match can not be made then the out case is executed causing an explicit failure.

operate(OPERATOR *operator*, OPARGS *operands*)

operate is a means of creating new values from other values, the *operands*. The effect of operate is highly dependent on *operator*. For example the operator '+' is different from the operator 'generate vector'. Each operator has a set of operands on which it can operate,

'+', for example, can only operate on numerical types and the ability for an operator to operate on a value is usually modal, '+' can operate on two ranges, but not vectors. Errors caused by the application of operators to values of the wrong mode can usually be detected before execution, however errors such as division by zero, in most cases, cannot be detected until execution, when an exception is delivered. The *operands* are executed in an order dependent on the particular operator,but always in the environment in which *operate* is executed, and the value delivered by *operate* is dependent on the values delivered by *operands*.

solve(SOLVE_CLAUSES *solve_clauses*)

solve is one of the ways of repeatedly executing code and is intended to model labels and gotos. This is not the only construction for handling flow through a program, there are also constructions designed for the representation of 'for constructions', 'loops' and 'conditionals'. These could be implemented in terms of solve, and are described in the Appendix. *solve* is very general and consequently can be difficult to analyse. It is the requirements of languages such as Algol68, Pascal and Ada to have a completely general flow control, and the ability to express some programs more easily in this manner, that necessitates the need for a *solve* construction.

Each SOLVE_CLAUSE is constructed from a set of LABELs, a BOOL and a LOAD. All LOADs are executed in the same environment as *solve*. The natural progression of load execution is to start with the LOAD of the first limb and continue through the loads in order as long as the BOOL is false, in much the same manner as load_sequence does. If the BOOL belonging to the clause is true, then the result of the clause is the result of the whole, and evaluation of the solve is complete. As with load_sequence, all LOADs, except for the last, must deliver void unless their BOOL is true. The last clause is always treated as if its BOOL is true. This progression is interrupted when an exception is delivered. If the exception corresponds to one of the LABELs in *solve_clauses* (see assert), then the load of the clause in which it lies is executed and the progression continues from there. There cannot be a duplication of any of the labels in the limbs as this would lead to an ambiguity when determining which load to execute. Either the program does not end, or eventually a value is delivered.

assert(ASSERTION *assertion*, LABEL_PROCS *jumps*, ASSARGS *operands*)

assert is a means of determining the flow of a program by the determination of some property of the values given by *operands*: for example, asserting whether a boolean value is true. If the assertion about the value holds then a value is delivered dependent on the assertion. Otherwise one of the *jumps* is selected, giving *jump*. If *jump* is a procedure then it must be one that always delivers an exception, and this is obeyed. If *jump* itself is an exception, then this is

delivered, and if it is a label, it is delivered as an exception to be trapped by an enclosing solve clause.

5. REFERENCES

1 Currie I.F., Edwards P.W. and Foster J.M.
"Flex firmware" RSRE Report No. 81009 (1981)

2 Currie I.F., Edwards P.W. and Foster J.M.
"PerqFlex firmware" RSRE Report No. 85015
(1981)

3 Goguen J.A., Thatcher J.W., Wagner E.G. and Wright J.B.
"Some fundamentals of order algebra semantics"
Proc. Symp. on Mathematical foundations of Computer Science 1976
pp 153-168

4 P.J.Landin
"The mechanical evaluation of expressions"
Computer Journal Vol 6 No 4 pp 308 - 320 Jan 1964

APPENDIX A: THE TEN15 SIGNATURE

Sorts

LOAD	Statements, expressions, program
INTEGER	
BOOL	
STRING	
MODE	
TUPLELOADS	Group of loads for n-tuple
SEQLOADS	Group of loads for sequence
NAME	Tag for naming a value
INTRODUCTION	Pair of a tag and a string
LABEL	Label names
SOLVE_CLAUSES	Labelled statements
FALOADS	Loads for a group of vectors or arrays
CASE_LIMBS	Branches of a case integer statement
UNION_LIMBS	Branches of a case union statement
MCLOADS	Loads for a group of union values
MULT_UNION_LIMBS	Branches of a multiple case union
MODEDS	Loads for a group of moded values
MODED_LIMBS	Branches of a case moded
OPERATOR	
OPARGS	Loads for arguments of operator
ASSERTION	
LABEL_PROCS	Labels, loads for procedures or fail numbers
ASSARGS	Loads for arguments of assertion
COMMENT	A comment (form not chosen yet)
LIMIT	Pair of integers for use in CASE
INTROTAGS	Group of pairs of introduction and union tags
INTROMODES	Group of pairs of introduction and mode
LABELS	
LIMITS	
INTEGERS	
MODES	

Arities

load_void: LOAD
illegal_load: LOAD
load_name: NAME $\longrightarrow$ LOAD
load_integer: INTEGER $\longrightarrow$ LOAD
load_boolean: BOOLEAN $\longrightarrow$ LOAD
load_sequence: SEQLOADS $\longrightarrow$ LOAD
load_tuple: TUPLELOADS $\longrightarrow$ LOAD
identity_declaration: INTRODUCTION $\times$ LOAD $\times$ LOAD $\longrightarrow$ LOAD
variable_declaration: INTRODUCTION $\times$ LOAD $\times$ LOAD $\longrightarrow$ LOAD
conditional: LABEL $\times$ LOAD $\times$ LOAD $\longrightarrow$ LOAD

```
loop: LABEL × LOAD ⟶ LOAD
solve: SOLVE_CLAUSES ⟶ LOAD
operate: OPERATOR × OPARGS ⟶ LOAD
assert: ASSERTION × LABEL_PROCS × ASSARGS ⟶ LOAD
commented_load: COMMENT × LOAD ⟶ LOAD
for: LABEL × LABEL × INTRODUCTION × LOAD × LOAD × LOAD × LOAD × LOAD
          ⟶ LOAD
forall: INTRODUCTION × FALOADS × LABEL × LABEL × LOAD × LOAD
          ⟶ LOAD
case_integer: LOAD × CASE_LIMBS × LOAD ⟶ LOAD
case_union: LOAD × UNION_LIMBS × LOAD ⟶ LOAD
multiple_case_union: MCLOADS × MULT_UNION_LIMBS × LOAD ⟶ LOAD
case_moded: MODEDS × MODED_LIMBS × LOAD ⟶ LOAD
marks: INTEGERS × LOAD ⟶ LOAD

no_seqloads: SEQLOADS
add_seqload: SEQLOADS × LOAD ⟶ SEQLOADS

no_tupleloads: TUPLELOADS
add_tupleload: TUPLELOADS × LOAD ⟶ TUPLELOADS

no_solve_clauses: SOLVE_CLAUSES
add_solve_clause: SOLVE_CLAUSES × LABELS × BOOL ×
                        LOAD ⟶ SOLVE_CLAUSES

one_faload: INTRODUCTION × LOAD ⟶ FALOADS
add_faload: FALOADS × INTRODUCTION × LOAD ⟶ FALOADS

no_case_limbs: CASE_LIMBS
add_case_limb: CASE_LIMBS × LIMITS × LOAD ⟶ CASE_LIMBS

no_union_limbs: UNION_LIMBS
add_union_limb: UNION_LIMBS × INTRODUCTION × INT ×
                      LOAD ⟶ UNION_LIMBS

one_mcload: LOAD ⟶ MCLOADS
add_mcload: MCLOADS × LOAD ⟶ MCLOADS

no_mult_union_limbs: MULT_UNION_LIMBS
add_mult_union_limb: MULT_UNION_LIMBS × INTROTAGS ×
                           LOAD  ⟶ MULT_UNION_LIMBS

one_moded: LOAD ⟶ MODEDS
add_moded: MODEDS × LOAD ⟶ MODEDS

no_moded_limbs: MODED_LIMBS
add_moded_limb: MODED_LIMBS × MODES × INTROMODES ×
                        LOAD ⟶ MODED_LIMBS

no_opargs: OPARGS
```

add_oparg: OPARGS × LOAD ⟶ OPARGS

no_assargs: ASSARGS
add_assarg: ASSARGS × LOAD ⟶ ASSARGS

no_label_procs: LABEL_PROCS
add_label_to_lp: LABEL_PROCS × LABEL ⟶ LABEL_PROCS
add_proc_to_lp: LABEL_PROCS × LOAD ⟶ LABEL_PROCS
add_fail_no_to_lp: LABEL_PROCS × INTEGER ⟶ LABEL_PROCS

make_introduction: NAME × STRING ⟶ INTRODUCTION

one_introtag: INTRODUCTION × INTEGER ⟶ INTROTAGS
add_introtag: INTROTAGS × INTRODUCTION × INTEGER ⟶ INTROTAGS

one_intromode: INTRODUCTION × MODE ⟶ INTROMODES
add_intromode: INTROMODES × INTRODUCTION × MODE
⟶ INTROMODES

no_labels: LABELS
add_label: LABELS × LABEL ⟶ LABELS

no_limits: LIMITS
add_limit: LIMITS × INTEGER × INTEGER ⟶ LIMITS

no_integers: INTEGERS
add_integer: INTEGERS × INTEGER ⟶ INTEGERS

no_modes: MODES
add_mode: MODES × MODE ⟶ MODES

APPENDIX B: LOAD CONSTRUCTORS IN THE PROGRAM ALGEBRA

This appendix is a summary of the ways of constructing LOADs in the Ten15 program algebra, and the effect of evaluating them.

illegal_load

This causes a standard exception.

load_void

This delivers the void value.

load_integer(INTEGER *i*)

This delivers the only value with mode range(*i*, *i*).

load_boolean(BOOL *b*)

This delivers either true, or false depending on *b*.

load_name(NAME *n*)

This delivers the value associated with the name in the environment.

load_sequence(SEQLOADS *sequence*)

Each of the loads in *sequence* is executed in order and all but the last must deliver the void value. The value delivered by the whole is the value delivered by the last load.

load_tuple(TUPLELOADS *tuple*)

Each of the loads in *tuple* is executed in order, and the n-tuple of the values delivered by each of these loads is delivered as the value of load_tuple. The type of the whole is the Cartesian product of the types of the parts.

identity_declaration(INTRODUCTION *n*, LOAD *value*, *scope*)

n is associated with the value delivered by *value*, and then added to the environment in which *scope* is executed.

variable_declaration(INTRODUCTION *n*, LOAD *value*, *scope*)

This is similar to identity_declaration except that a reference is generated for the value delivered by *value* and it is this which is associated with *n* in the execution of *scope*.

conditional(LABEL *l*, LOAD *cond*, *else*)

cond is executed: if the value delivered is not the exception corresponding to *l* then it is delivered, otherwise *else* is executed.

loop(LABEL *l*, LOAD *loop*)

loop is repeatedly executed until a value which is not the exception corresponding to *l* is delivered. This is the value delivered by loop.

solve(SOLVE_CLAUSES *clauses*)

A SOLVE_CLAUSE consists of a set of LABELs, a BOOL and a LOAD. If the BOOL is true, the result of that clause is the result of the whole. The BOOL of the last clause is deemed to be true. The loads of each clause are executed in turn until an exception occurs which is not for one of

the labels in the solve, or a clause is evaluated with a true BOOL. At this point this exception or result is delivered. If the BOOL for the clause is false, execution continues with the next clause, and the result of that clause must be void. If the exception is for one of the labels in the solve, execution continues with the labelled clause.

for(LABEL *step, end*,INTRODUCTION *loopid*,LOAD *start, by, to, load, endl*)

to, *start* and *by* are executed. They must all deliver ranges. The integer delivered by *start* is associated with *loopid* and its mode is the smallest range containing all the possible integers a value could take in moveing from *start* to *to* in steps of *by*. *load* is then evaluated in the context of this environment. If the value delivered is not an exception corresponding to either *step* or *end*, then it is delivered, if *endl* is not a load then this must be the void value. If the value delivered is an exception corresponding to *step* then the value associated with *loopid* is increased by *by*. If this value is now beyond *to*, then *endl* is executed in the original environment, that is without *loopid*;if the value is not beyond *to*, then *load* is executed again with the new value association for *loopid*. If the value delivered is an exception corresponding to *end*, then *endl* is executed.

forall(NAME *loopid*, FALOADS *vec_array*,
LABEL *step, end*, LOAD *load, endl*)

forall is similar to for, except that the value associated with *loopid* is determined by a set of vectors or arrays or a combination of both. The FALOADS, *vec_array*, consists of a set of pairs of INTRODUCTIONs and LOADs. The loads in *vec_array* are executed in turn. Each one must deliver either a one dimensional array or a vector; further, all vectors and arrays must be the same size. Before *load* is executed for the first time, *loopid* is associated with the integer 1 and its mode is deduced from the ranges of the bounds of the vectors and arrays. Each name introduced in the INTRODUCTIONs in *vec_array* is associated with the first reference of the corresponding vector or array (In the case of array the first reference is given by the lower bound) . If the load delivers the label corresponding to *step* then the value associated with *loopid* is increased by 1. If this is larger than the size of the vectors or arrays, *endl* is obeyed , otherwise each name is associated with the next reference in the vec/array and *load* is re-executed.

case_integer(LOAD *control*, CASE_LIMBS *limbs*, LOAD *out*)

A CASE_LIMB consists of a list of INTEGER ranges and a LOAD. *control* is obeyed, and the value delivered must be a range. Each of the limbs in *limbs* is examined in order, until a limb is found where the value delivered by *control* lies in one of its range. The LOAD in the limb is then obeyed. If there is no such limb, then *out* is obeyed.

case_union(LOAD *union*, UNION_LIMBS *limbs*, LOAD *out*)

A UNION_LIMB consists of an INTEGER tag, an INTRODUCTION and a LOAD. *union* is obeyed, and the value delivered must be a union. Each of the limbs in *limbs* is examined in order, until a limb is found where the union delivered by *union* has the same tag as the one specified in the limb. The value in the field of the union is then associated with the name in the limb and the load of the limb is obeyed in an environment which has this value - name association added to it. If there is no such limb, then *out* is obeyed.

multiple_case_union(MC union, MULT_UNION_LIMBS limbs, LOAD *out*)

multiple_case_union is a generalisation of case_union. Instead of matching the tag of one union with a limb, associating a name with the field of the union and obeying the load in this context, it matches the tags of all the unions in *union* with all of a list of INTEGER tags in the limb, associates the corresponding field of each union with a name from a list of INTRODUCTIONs in the limb and obeys the load of the limb in the context of this new environment. A MULT_UNION_LIMB consists of a set of pairs of INTRODUCTIONs and INTEGER tags; and a LOAD, the load to be obeyed.

case_moded(MODEDS *modeds*, MODED_LIMBS *limbs*, LOAD *out*)

case_moded is similar to other case constructors in that it attempts to match some incoming values, in this case modeds, with some of their possible properties and on the basis of this evaluate a load in which some part of the incoming values is made available to it. With case_moded each limb is testing to determine mode relationships of the modes of the incoming values, and makes the value of the moded available inside the load of the limb. A MODED_LIMB is constructed from a list of formal modes; a list of pairs of an INTRODUCTION and a MODE (written in terms of the formal modes) to associate with the values in the *modeds*; and a LOAD to be evaluated if a match is made.

operate(OPERATOR *operator*, OPARGS *operands*)

operate obeys all of the operands in an order dependent on the individual operator. Then a value is delivered dependent on the values delivered by the operands and the operation.

assert(ASSERTION *assertion*, LABEL _PROCS *labs*, ASSARGS *operands*)

assert obeys all of the operands in an order dependent on the individual assertion. Then if the values delivered by the operands have a certain property, dependent on the ASSERTION, (e.g. Whether a boolean value is TRUE) then a value, determined by the assertion and the operand values,

is delivered; otherwise, one of the labs is selected. If it is a LOAD delivering a procedure which when obeyed causes an exception, it is obeyed, if it is a trap value then a failure exception is delivered and if it is a label a label exception is delivered.

commented_load(COMMENT *c* , LOAD *l*)

This for associating a comment with a load.

APPENDIX C: COERCION

The range of any value in Ten15 is taken to be the smallest possible one which can be deduced. This gives the most information to the translator, which can avoid the introduction of unnecessary checks. However, the reverse is true of procedure parameters and references. Here we normally want to define a range as wide as is reasonable, in order that the procedure should be applicable to as many values as possible. So when we are faced with the application of a procedure to a range parameter, we are likely to find that the parameter has a smaller range than that specified in procedure parameter. There is an operator which changes the range of a value to a specified range. If the new range includes the range of the value and occupies the same size, a translator will not have to provide any code to effect this change. If the new range is smaller, code will have to be provided to check that the value lies in the range and give an exception otherwise. so we could overcome the difficulty about proicedure parameters by putting a change range operation in, which will usually not generate any code.

This would be a perfectly viable solution, but a change range operation would be necessary in almost every procedure application and assignment to reference. So we have chosen to make this change range operation implicit: the range of a value can always be widened to fit such requirements. This may involve changing the size occupied by the data, if it goes from single to double length.

There are other similar places. In a conditional, the overall value is delivered from one of the two branches. If this result is a range, the values in the two branches are likely to have different ranges, again because we are keeping them as narrow as possible. The translator could insert change range operations, so that we can give the two results the same mode, but we have chosen to make this implicit. This also has the effect that the result of the overall conditional has the smallest range including the ranges of its branches. Similar remarks apply to the case constructions.

The rule, then, is that implicit change range operations will be applied to range values in these situations to make the values fit. Similar changes are made to real numbers. But no other coercions are provided.

APPENDIX D: OPERATORS IN TEN15

This is a list giving a brief description of all the operators in Ten15. In the case where the operator is shown to have a parameter, this indicates that the operator is dependent on some value in the Ten15 program algebra, that is some value which is known at translation as opposed to execution.

In all operations delivering values whose mode is constructed from range, unless explicitly stated, the range of the result is the smallest range containing all possible results, given the modes of the operands. There are three types of arithmetic on ranges, those that overflow when the result can not be represented in one word, those that overflow if the result cannot be represented in two words, and those that do not overflow. It is appreciated that the first two forms are machine dependent, but machine independence can be obtained from using the latter. The first two forms are included for convenience and the arithmetic can be thought of as parameterised by the word size of the machine.

In all operations delivering values whose mode is constructed from real, unless otherwise stated, the mode of the result is determined by the real operands of the operator. The mantissa of the result mode is the maximum of the mantissas of the modes of the real operands and the exponent of the result mode is the maximum of the exponents of the modes of the real operands. If it is not possible to express the result with a value of such a mode, then an exception is delivered.

There are some modes which have read_only parallels, for example vector, array, reference; unless explicitly stated, all operators operating on such modes are assumed to also operate on the corresponding read_only mode; in this case, where the value delivered is such a mode it inherits the read only property from the operand.

On all operations that require the generation of space, even though the description may say that there are no exceptions, in practice there may space limitations on the machine causing a 'store full' exception.

1. one_word_plus

This operates on two values whose modes are constructed from range. Either their sum is delivered or, if there is a numeric overflow, an exception. The overflow occurs if the result is too large for a one word representation.

2. one_word_minus

This operates on two values whose modes are constructed from range. Either their difference is delivered or, if there is a numeric overflow, an exception. The overflow occurs if the result is too large for a one word representation.

3. one_word_multiplication

This operates on two values whose modes are constructed from range. Either their product is delivered or, if there is a numeric overflow, an exception. The overflow occurs if the result is too large for a one word representation.

4. one_word_division

This operates on two values whose modes are constructed from range. Either their quotient is delivered or, if there is a numeric overflow, an exception. The overflow occurs if the result is too large for a one word representation.

5. maximum

This operates on two values whose modes are constructed from range. Their maximum is delivered and there is no overflow.

6. minimum

This operates on two values whose modes are constructed from range. Their minimum is delivered and there is no overflow.

7. identity

This is the identity operation, operating on any value.

8. one_word_negate

This operates on a value whose mode is constructed from range. Either its negation is delivered or, if there is a numeric overflow, an exception. The overflow occurs if the result is too large for a one word representation.

9. one_word_abs

This operates on a value whose mode is constructed from range. Either its absolute value is delivered or, if there is a numeric overflow, an exception. The overflow occurs if the result is too large for a one word representation.

10. change_range(*mode*)

This operator changes the range of either an integer range or a real range. *mode* must be either an integer range or a real range. It operates on a single operand, if this operand is an integer range it converts it to *mode*, making any checks that are necessary to insure that the convertion is possible. If these checks fail, then an exception is produced. If the operand is a real range, then it can be converted to another real range, again with the necessary checks being made.

11. no_overflow_plus

This operates on two values whose modes are constructed from range. Their sum is delivered with no exception.

12. no_overflow_minus

This operates on two values whose modes are constructed from range. Their difference is delivered with no exception.

13. no_overflow_multiplication

This operates on two values whose modes are constructed from range. Their product is delivered with no exception.

14. no_overflow_modulo/division

This operates on two values whose modes are constructed from range. The range of the denominator must be positive. The Cartesian product of the remainder and the quoitient are delivered, with no exception, the remainder being non-negative.

15. entier(*range*)

range is a mode which must be an integer range. This operates on one operand which must be a real range and delivers an integer with mode *range*, the truncation of the real. If the integer delivered does not lie in the range *range*, then an exception is produced.

16. round(*range*)

range is a mode must be an integer range. This operates on one operand which must be a real range and delivers an integer with mode *range*, the nearest integer to the real. If the integer delivered does not lie in the range *range*, then an exception is produced.

17. real_power_int

This operates on a real range and an integer range, raising the real to the power of the integer. An exception is delivered in the case of overflow.

18. odd

This operates on any integer range delivering a boolean. There are no exceptions.

19. two_word_plus

This operates on two values whose modes are constructed from range. Either their sum is delivered or, if there is a numeric overflow, an exception. The overflow occurs if the result is too large for a two word representation.

20. two_word_minus

This operates on two values whose modes are constructed from range. Either their difference is delivered or, if there is a numeric overflow, an exception. The overflow occurs if the result is too large for a two word representation.

21. two_word_multiplication

This operates on two values whose modes are constructed from range. Either their product is delivered or, if there is a numeric overflow, an exception. The overflow occurs if the result is too large for a two word representation.

22. two_word_division

This operates on two values whose mode is constructed from range. Either their quotient is delivered or, if there is a numeric overflow, an exception. The overflow occurs if the result is too large for a two word representation.

23. two_word_negate

This operates on a value whose modes are constructed from range. Either its negation is delivered or, if there is a numeric overflow, an exception. The overflow occurs if the result is too large for a two word representation.

24. two_word_abs

This operates on a value whose modes are constructed from range. Either its absolute value is delivered or, if there is a numeric overflow, an exception. The overflow occurs if the result is too large for a two word representation.

25. no_overflow_division

This operates on two values whose modes are constructed from range. The range of the denominator must not contain 0. The quoitient is delivered, with no exception.

26. no_overflow_negate

This operates on a value whose mode is an integer range. Its negation is delivered with no exception.

27. no_overflow_abs

This operates on a value whose mode is constructed from range. Either its absolute value is delivered or, if there is a numeric overflow, an exception. The overflow occurs if the result is too large for a two word representation.

28. real_plus

This operates between two real ranges, delivering their sum with a possible overflow exception.

29. real_minus

This operates between two real ranges, delivering their difference with a possible overflow exception.

30. real_multiplication
This operates between two real ranges, delivering their product with a possible overflow exception.

31. real_division
This operates between two real ranges, delivering their quotient with a possible overflow exception.

32. real_abs
This operates on a real range, delivering its absolute value with a possible overflow exception.

33. real_negate
This operates on a real range, delivering its negation with a possible overflow exception.

34. no_overflow_modulo
This operates on two values whose modes are constructed from range. The range of the denominator must be positive. The remainder, which is non-negative, is delivered, with no exception.

36. real_maximum
This operates between two real ranges, the maximum is delivered with no exception.

37. real_minimum
This operates between two real ranges, the minimum is delivered with no exception.

38. vector_content_equality
This operates on two vectors. It delivers true if their contents are equal. An exception is produced if the vectors are of differing size.

39. vector_content_inequality
This operates on two vectors. It delivers true if their contents are not equal. An exception is produced if the vectors are of differing size.

40. equality
This operates on two ranges or two other values of the same mode, delivering true if they are equal. There are no exceptions.

41. inequality
This operates on two ranges or two other values of the same mode, delivering true if they are unequal. There are no exceptions.

42. greater_than
This operates on two integer ranges, delivering true if the first is greater than the second. There are no exceptions.

43. greater_than_or_equal_to
This operates on two integer ranges, delivering true if the first is greater than or equal to the second. There are no exceptions.

44. less_than
This operates on two integer ranges, delivering true if the first is less than the second. There are no exceptions.

45. less_than_or_equal_to
This operates on two integer ranges, delivering true if the first is less than or equal to the second. There are no exceptions.

46. static_range_check(*range*)
This operates on an integer range, delivering true if the value lies in *range*. There are no exceptions.

47. dynamic_range_check
This operates on three integer ranges, delivering the value of the first operand if it lies between the second and third; otherwise an exception is delivered.

48. and
This operates on two booleans, delivering a boolean; there are no exceptions.

49. or
This operates on two booleans, delivering a boolean; there are no exceptions.

50. xor
This operates on two booleans, delivering a boolean; there are no exceptions.

51. not
This operates on a boolean, delivering a boolean; there are no exceptions.

52. real_greater_than
This operates on two real ranges, delivering true if the first is greater than the second. There are no exceptions.

53. real_greater_than_or_equal_to

This operates on two real ranges, delivering true if the first is greater than or equal to the second. There are no exceptions.

54. real_less_than

This operates on two real ranges, delivering true if the first is less than the second. There are no exceptions.

55. real_less_than_or_equal_to

This operates on two real ranges, delivering true if the first is less than or equal to the second. There are no exceptions.

56. dynamic_upb_array

This takes two operands, the first of which is an array and the second an integer range. The second operand specifies the dimension of the array, and its upper bound is delivered. An exception is delivered if the second parameter does not lie between 1 and the dimension of the array.

57. dynamic_lwb_array

This takes two operands, the first of which is an array and the second an integer range. The second operand specifies the dimension of the array, and its lower bound is delivered. An exception is delivered if the second parameter does not lie between 1 and the dimension of the array.

60. index_vector

This operates on a vector and an integer range. The vector is indexed by the range, delivering a reference.

61. trim_above

This operates on a vector and an integer range. The vector referencing the set of values above the one indexed by the integer is delivered. There are no exceptions.

62. trim_upwards_from

This operates on a vector and an integer range. The vector referencing the set of values above and including the one indexed by the integer is delivered. There are no exceptions.

63. trim_below

This operates on a vector and an integer range. The vector referencing the set of values below the one indexed by the integer is delivered. There are no exceptions.

64. trim_downwards_from

This operates on a vector and an integer range. The vector referencing the set of values below and including the one indexed by the integer is delivered. There are no exceptions.

65. index_array

The number of operands of this operator depends on the dimension of the array. There are one greater than the dimension of the array. The first must be an array, the remainder are integer ranges used to index the array. the value delivered is a reference. An exception is delivered if any of the integers are out of the bounds of their respective dimensions.

66. trim/slice array

The number of operands of this operator depends on the dimension of the array; there are one greater than the dimension of the array. The first must be an array, the remainder are trimscripts, one for each dimension. Trimscripts consist of either void, which does not alter the dimension, an integer range, which slices the array in that dimension, or a tuple of three ranges causing a trim in the dimension of the form [i:j AT k] where the tuple is (i,j,k). If the number of slices is equal to the dimension of the array, a reference is delivered; otherwise an array is delivered. An exception is delivered if the indices are outside the bounds of their dimension.

67. upb_array(*dim*)

This takes one operand, which must be an array. It delivers the upper bound of dimension *dim*. There are no exceptions.

68. lwb_array(*dim*)

This takes one operand, which must be an array. It delivers the lower bound of dimension *dim*. There are no exceptions.

70. vector_size

This takes one operand which must be a vector. The size of the vector is delivered. There are no exceptions.

71. select_from_struct(*field*)

field is an integer identifying a component of a structure. There is one operand, which must be a structure and the component *field* of the structure is delivered. There are no exceptions.

72. select_from_ref/ptr_struct(*field*)

field is an integer identifying a component of a structure. There is one operand, which must be a reference or pointer to a structure and a reference to the component *field* of the structure is delivered. There are no exceptions.

73. replace_field(*field*)

field is an integer identifying a component of a structure. There are two operands, a structure and a value with a mode the same as the component *field* of the structure. The value delivered is a structure of the first operand with *field* replaced by the second operand. There are no exceptions.

74. select_from_vec/array_struct(*field*)

field is an integer identifying a component of a structure. There is one operand, which must be a vector/array to a structure and an array referencing the component *field* of the structure is delivered. There are no exceptions.

80. assign_to_pointer

This has two operands, a pointer which cannot be read_only and a value with a mode the same as that from which the pointer is constructed. The value is assigned to the pointer and void is delivered. There are no exceptions.

81. assign_to_reference

This has two operands, a reference which cannot be read_only and a value with a mode the same as that from which the reference is constructed. The value is assigned to the reference and void is delivered. There are no exceptions.

82. assign_to_vector

This has two operands, a vector which cannot be read_only and another vector of the same mode which can either be read_only or not. The contents of the second operand are assigned to the first and void is delivered. If the vectors are not of the same size, an exception is delivered.

83. generate_pointer

This has one operand which can be of any mode. A pointer to the operand is generated and delivered. There are no exceptions.

84. pack_to_vector

This has one operand which is a structure of values of the same mode. A vector containing the components of the structure is generated and delivered. There are no exceptions.

85. generate vector

This has two operands. The first is an integer range and the second a value of any mode. A vector is generated, containing the value of the first operand copies of the second operand, and is delivered. There are no exceptions.

86. generate_array

The number of operands of this operator is one greater than twice the dimension of the array to be generated. The first operand can be of any mode and the remaining operands must be integer ranges. The integer ranges are taken in pairs and they specify the bounds of each dimension of the array. An array of this size is then generated and filled with copies of the first operand. The array is delivered. An exception is delivered if any of the values of the ranges give unsuitable bounds, that is if the lower bound is more than one greater than the upper bound for any dimension.

87. de_pointer

This operates on a pointer and delivers the value pointed at by it. There are no exceptions.

88. de_reference

This operates on a reference and delivers the value referenced by it. There are no exceptions.

89. pointer_to_reference

This operates on a pointer to a value and delivers a reference to the same memory space. There are no exceptions.

90. to_moded

This operates on a value of any mode and delivers its moded form, that is the value bound with its mode. There are no exceptions.

91. unite_to_mode(*mode*,*tag*)

mode is a union and *tag* identifies one of its fields. The operand is a value with a mode identical to the tagged field. This value is united to the mode *mode* and delivered. There are no exceptions.

92. vector_to_array

The number of operands of this operator is one greater than twice the dimension of the array to be generated. The first operand is a vector and the remaining operands must be integer ranges. The integer ranges are taken in pairs and they specify the bounds of each dimension of the array. An array of this size is then created accessing the same memory space as the vector. The array is delivered. An exception is delivered if the size of the array specified by the bounds is not equal to the size of the vector or if any of the values of the ranges give unsuitable bounds, that is if the lower bound is more than one greater than the upper bound for any dimension.

93. array_to_vector

This operates on an array and delivers a vector of the same size as the array accessing the same memory space. This produces an exception if the array is not contiguous.

94. is_in_union_field(*tag*)
This operates on a union and delivers the boolean true if the field of the union is *tag*. There are no exceptions.

95. boolean_dynamic_range_check
This operates on three integer ranges delivering the boolean true iff the first operand lies between the second and third. There are no exceptions.

97. discard
This operates on a value of any mode and delivers void. There are no exceptions.

100. apply
This operates on a procedure and its parameters. It applies the procedure to its parameters and delivers the result.

101. trap_apply
This operates on a procedure and its parameters. It applies the procedure to its parameters and delivers a union which is either the result or a trapped exception. There are no exceptions.

102. close
This operates on the non-local to a package of order one and the package to deliver its closure as a procedure. There are no exceptions.

103. trap_to_moded
This operates on a trap value and delivers the moded value from which it was created - see fail. There are no exceptions.

104. close_package
This operates on the non-local to a package and the package to deliver its closure as a structure of procedures. There are no exceptions.

105. close_recursive
This operates on the non-local to a recursive and the recursive to deliver its closure as a structure of procedures. There are no exceptions.

106. close protocode
This operates on the constants of a protocode and the protocode to deliver its closure as a package. There are no exceptions.

110. fail
This operates on a value of any mode. If it is a trap value then the exception of the trap is delivered, otherwise the value is moded, made into a trap value and its exception is delivered.

111. launch_process

This operates on a procedure and its parameters. It launches the procedure applied to its parameters as a process. A structure of two procedures is delivered. The first is a proc(moded->( )) which will cause the process to abort with a trap value formed from the moded parameter supplied. The second is a proc( )->union(result,trap,( )), which delivers the trap if the process has failed else if has not finished then void, else the result .

112. create empty_vector

This creates an empty vector of an unspecified mode. There are no exceptions.

113. generate_value(*mode*)

This generates a value of mode *mode*.

114. to_choice

This operates on any value to deliver the non-void choice of that value. There are no exceptions.

115. am_i_me

This operates on void to deliver a procedure(void,bool) which when called delivers true iff it is obeyed in the same process as the one in which this operator was obeyed. There are no exceptions.

116. make_unabortableable

This operates on a procedure to deliver a procedure of the same mode which cannot be aborted using the abort procedure delivered by launch process.

117. permit_abortable

This operates on a procedure and its parameters in the same way as trap_apply. It allows an abortable procedure to be called from within an unabortable procedure, trapping resulting failures.

118. make_barrier(*mode*)

This operates on a boolean value, the position of the barrier, and delivers a structure of three procedures. The first and last are of mode proc(void,void) for passing and raising the barrier; the second is a proc(*mode*,bool). *mode* is a non-negative range, When this procedure is called with the value specifies a time, some time after which the queue at the barrier is left. The boolean value delivered is dependent on whether the barrier was passed or not.

119. make_generic_table(*mode*)

This is an operater which delivers two procedures a finder and a table constructor. The table can be thought of as a set of pairs of values and

the finder takes a value searches through the table and attempts to find a pair whose first element is the same as the parameter of the finder. If such a pair is found, the second element of the pair is delivered as the first field of a union; otherwise the second field of the union is delivered, which is void. The finder delivered by the operator delivers the second element of the union with any parameter. The table constructor takes a pair as a parameter and delivers a new table constructor and finder for adding a pair to the new table and finding elements in the new table. *mode* determines the form of the values that can be added to the table.

128. anything_to_words

This is a machine dependent operation which takes any value and delivers a structure consisting of the representing words.

121. persist

This operates on a value and makes a persistent value holding the value.

122. unpersist

This operates on a persistent value and makes a copy of the value held by it. There are no exceptions.

123. make_persistent_variable

This operates on a persistent value making a reference to it. There are no exceptions.

124. make_void_choice(*mode*)

This makes the void choice of a *mode*. There are no exceptions.

125. make_read_only

This operates on values with read_only parallels and delivers the read_only value accessing the same memory. There are no exceptions.

126. delay

This operates on a numeric value and causes a delay in the process in which the operation is obeyed by at least the value of its operand.

127. assign_to_persistent_variable

This operates on a persistent_variable and a persistent value, assigning the value to the variable and delivering void. There are no exceptions.

128. deref_persistent_variable

This operates on a persistent variable to deliver the persistent value referenced by it. There are no exceptions.

129. cycle array dimensions(*i1, i2, . . , in*)

This operates on an array. *n* is the dimension of the array and

(i1, i2, .. in) is a permutation of the integers 1..*n*. The dimensions of the array are permuted corresponding to this permutation and the new array is delivered.

130. abs on bool

This operates on a boolean value. A range(0,1) is delivered. 1 is delivered if the value is true otherwise) is delivered.

131. generate block

This has two operands. The first is an integer range and the second a value of any mode. A block is generated, containing the value of the first operand copies of the second operand, and is delivered. There are no exceptions.

132. pack_to_block

This has one operand which is a structure of values of the same mode. A block containing the components of the structure is generated and delivered. There are no exceptions.

133. andth

This has two operands which must be booleans. If the first is true then the second is evaluated and delivered, otherwise it is not evaluated. There are no exceptions.

134. orel

This has two operands which must be booleans. If the first is false then the second is evaluated and delivered, otherwise it is not evaluated. There are no exceptions.

135. block_to_vec

This operates on a block of values and delivers a vector referencing the values in the block.

136. assign_to_array

This has two operands, an array which cannot be read_only and another array of the same mode which can either be read_only or not. The contents of the second operand are assigned to the first and void is delivered. If the arrays do not have the same bounds, an exception is delivered.

137. increase_array_dimensionality(*dim, bound*)

This operates on an array and adds an extra dimension with one index. *dim* is the new dimension which must lie between one and one greater than the old dimension and *bound* is the lower and upper bound of that dimension. There are no exceptions.

APPENDIX E: ASSERTIONS IN TEN15

This list is a brief description of all the assertions in Ten15. In the case where the assertion is shown to have a parameter, this indicates that the operator is dependent on some value in the Ten15 program algebra, that is some value which is known at translation as opposed to execution.

Assertions always require a LABEL_PROC. Assertions assert a property of their operands. If the assertion is correct then a value is delivered: if the assertion is incorrect then depending on the LABEL_PROC either a failure exception is delivered, a label exception is delivered or a procedure delivering an exception is called.

1. jump

This is an assertion which is never correct, it operates on void.

2. is_false

This operates on a boolean value, asserting that it is false and delivering void.

3. is_true

This operates on a boolean value, asserting that it is true and delivering void.

4. is_in_union(*field*)

This operates on a union asserting that the tag of the union is *field*. The value delivered is the field of the union.

5. is_moded(*m*)

This operates on a moded value, asserting that its mode is *m*. The value delivered is the value bound in the moded.

6. in_range(*mode*)

mode is a mode which must be an integer range.This operates on a range and asserts that the operand lies in the range of *mode*. The value delivered is the operand with its mode changed to *mode*.

7. label_to_proc

This is an assertion which is always correct. It delivers a procedure which when obeyed causes the effect of having produced the exception in the assertion.

8. choice_not_void

This operates on a choice, asserting that it is not the void choice, delivering the non-void choice.

U.S. DEPARTMENT OF COMMERCE
National Technical Information Service
Alexandria, VA 22312 703-605-6000