Architectures, Languages and Techniques
B. M. Cook (Ed.)
IOS Press, 1999

User-Defined Data Types and Operators in occam

David C. WOOD and James MOORES
Computing Laboratory, University of Kent at Canterbury, CT2 7NF
{D.C.Wood, jm40}@ukc.ac.uk

Abstract: This paper describes the addition of user-defined monadic and dyadic operators to occam* [1], together with some libraries that demonstrate their use. It also discusses some techniques used in their implementation in KRoC [2] for a variety of target machines.

1. Introduction

Most programming languages allow user-defined data types, and many (Algol-68 [3], Ada [4]; Fortran-90 [5], etc.) also provide facilities for user-defined operators, acting on variables of both built-in and user-defined types.

A mechanism has been provided in KRoC to enable operators to be written by any user of occam.

The main advantage of these is computational convenience, avoiding the labour of writing expressions in terms of function calls. They have the additional advantage of being able to be overloaded on a range of different data types. For example, '+' already represents addition for all the standard occam numerical types, and can now also be used for user-defined types like complex numbers.

In KRoC, such operators can also provide access to hardware implementing useful operations and data types not directly represented in occam.

The operators described here have been implemented on a range of machines, including the SPARC [6], DEC Alpha [7], MIPS [8], Motorola 68000 [9], and Analog Devices SHARC [10]. These operators are summarized in the appendices.

1.1. User-Defined Data Types

User-defined data types were introduced in occam 2.1 (then called occam 2.5) [11], but operations on them were restricted to the operators inherited from the base types, or to functions.

Standard occam data types may be renamed:

```
DATA TYPE THIS IS INT:
```

* occam is a trademark of the SGS-THOMSON Microelectronics Group.

Such types inherit the operators of their base types, but they cannot be mixed in expressions.

Also, RECORDs may be defined:

```
DATA TYPE THAT
  RECORD
    INT which, what:
:
```

The elements of such records may be specified as follows:

```
THAT thing:
INT why, who:
SEQ
  thing[which] := why
  thing[what]  := who
```

or

```
  thing := [why, who]
```

Where necessary, constants of user-defined types can be decorated to make them unambiguous; for example

```
VAL where IS [27, 42] (THAT):
```

Variables of such types may be passed as parameters and results of FUNCTIONs, but this leads to clumsy forms of expressions:

```
THAT x, y, z:
SEQ
  z := THAT.PROD (THAT.SUM (x, y), THAT.DIFF (x, y))
```

1.2. User-Defined Operators

User-defined operators provide a much more concise and convenient notation:

```
  z := (x + y) * (x - y)
```

The INMOS / SGS-THOMSON occam 2.1 *Toolset* compiler has been modified to allow user-defined operator. Such operators are declared like normal occam FUNCTIONs, except that the function name is replaced by a string; for example:

```
THAT FUNCTION "+" (VAL THAT x, y) ...
```

Operator functions must have either one or two parameters, for monadic (prefix) and dyadic (infix) operators, respectively, and a single result.

Only a limited number of operator symbols can be used. The standard occam operators (except type conversions) can be overloaded on new data types. The existing operators can also be redefined on existing types*. A few new operator symbols are provided: ^, ++, ==, ??, !!, $$, %, %%, %>, <%, @, @@, @>, <@, &&, &>, <&, [>, <]. All of these can be used as both monadic and dyadic operators.

* This is normally a Very Bad Thing, but see Section 4.

There is no operator precedence in occam, and associativity is treated very strictly; so, for example, `x PLUS y PLUS z` is legal, but `x + y + z` is not (because overflow might occur in some interpretations and not others). Redefined operators inherit their associativity, and new operators are not associative.

Some operations are still most conveniently represented as functions.

2. Example Operators

The following examples show a number of applications of user-defined operators in occam. Their implementation is discussed in Section 3.

2.1. Powers

An obvious operator to provide is power, or exponentiation. Although not required very often, it is not trivial to implement efficiently, so it may be worth providing in a library. Many programming languages include it.

Powers of all sizes of integer and real operands have been defined; for example

```
INT    FUNCTION "^" (VAL INT    X,         Y) ...
INT64  FUNCTION "^" (VAL INT64  X, VAL INT Y) ...
```

(these require $Y \geq 0$), and

```
REAL32 FUNCTION "^" (VAL REAL32 X, VAL INT Y) ...
REAL64 FUNCTION "^" (VAL REAL64 X, VAL INT Y) ...
```

These power functions are implemented by the square-and-multiply algorithm, and so are reasonably efficient. This is especially important with operand types for which multiplication is expensive. For this reason, special squaring functions have been provided for those data types for which this can be done more efficiently than by simple multiplication.

Powers with real exponents have also been provided:

```
REAL32 FUNCTION "^" (VAL REAL32 X, Y) ...
REAL64 FUNCTION "^" (VAL REAL64 X, Y) ...
```

Powers of most of the numerical data types described below are implemented similarly.

2.2. Rotations

Rotations are an example of a useful operation often provided by hardware, which is not usually available from high-level languages.

Left and right rotation operators, `<@` and `@>`, have been defined for all integer types; for example:

```
INT   FUNCTION "<@" (VAL INT   X,         N) ...
INT   FUNCTION "@>" (VAL INT   X,         N) ...
INT64 FUNCTION "<@" (VAL INT64 X, VAL INT N) ...
INT64 FUNCTION "@>" (VAL INT64 X, VAL INT N) ...
```

The shift count is taken modulo the size of the operand.

Related operations might be signed shifts, shifts that allow negative counts, and shifts (scaling by powers of two) applied to floating-point variables.

2.3. Extended Arithmetic

Many machines have instructions to produce double-length products from two single-length operands, and to divide double-length by single-length operands, giving single-length quotient and remainder. Operators have been written to represent these.

A new operator symbol is required for multiplication, but division can be overloaded.

The following are available on the SPARC:

```
INT64 FUNCTION "@" (VAL INT   X,           Y) ...
INT   FUNCTION "/" (VAL INT64 X, VAL INT   Y) ...
INT   FUNCTION "\" (VAL INT64 X, VAL INT   Y) ...
```

and the following in the 68000:

```
INT   FUNCTION "@" (VAL INT16 X,           Y) ...
INT16 FUNCTION "/" (VAL INT   X, VAL INT16 Y) ...
INT16 FUNCTION "\" (VAL INT   X, VAL INT16 Y) ...
```

In each case, they correspond closely to the hardware instructions.

The MIPS ‘doubleword multiply’ instructions give an `INT128` (§ 2.6) product from `INT64` operands. This is also easy on the DEC Alpha. The compiler library multiplication function for `INT64` (`INT64MUL%CHK`) generates a 128-bit result internally (and hence `INT128` multiplication produces an `INT256`!).

Similar floating-point instructions, `fsmuld` and `fdmulq`, are defined in the SPARC architecture, though probably not implemented in hardware, giving:

```
REAL64  FUNCTION "@" (VAL REAL32 X, Y) ...
REAL128 FUNCTION "@" (VAL REAL64 X, Y) ...
```

These could be used, for example, in an inner-product operator between arrays, in situations where cancellation must be minimized:

```
REAL64 FUNCTION "@" (VAL []REAL32 X, Y)
  REAL64 Result:
  VALOF
    SEQ
      ASSERT ((SIZE X) = (SIZE Y))
      Result := 0.0
      SEQ i = 0 FOR SIZE X
        Result := Result + (X[i] @ Y[i])
    RESULT Result
:
```

(Note that operators can have open arrays as parameters, though not as results.)

Quotient and remainder are often required together, and much of the computation is common to both, so the following is provided on the SPARC

```
INT, INT FUNCTION INT64DIVREM32 (VAL INT64 X, VAL INT Y) ...
```

which is essentially the same as `LONGDIV`, and the following on the 68000

```
INT16, INT16 FUNCTION INT32DIVREM16
                                (VAL INT X, VAL INT16 Y) ...
```

2.4. Complex Numbers

Several new numerical types have been defined. Most of the operators provided for them are redefinitions of the familiar arithmetic operations.

Complex numbers are very important in scientific computing, and are provided as standard types in Fortran [12], Algol-68, etc.

The data types are declared as occam 2.1 `RECORD`s:

```
DATA TYPE COMPLEX32
  RECORD
    REAL32 real, imag:
:
```

and

```
DATA TYPE COMPLEX64
  RECORD
    REAL64 real, imag:
:
```

A `COMPLEX128` type is defined similarly (§ 2.8).

Operators or functions for constructing complex numbers from reals, and selecting the real and and imaginary parts of complex numbers, are not required:

```
REAL64 x, y:
COMPLEX64 z:
SEQ
  z := [x, y]
  ...
  x, y := z[real], z[imag]
```

All the meaningful standard occam operators are provided. (This includes the comparisons '`=`' and '`<>`', although they a numerically dubious. Other comparisons are, of course, meaningless, and it is difficult to think of an interpretation for '`\`'.) Powers (with integer exponents) are also implemented, and complex conjugate is represented by '~'.

The predefined dyadic operators in occam (with the exception of shifts) require that both operands should be of the same type. However, the the '`*`' and '`/`' operators have been overloaded for the following useful combinations of operands of compatible sizes: `COMPLEX*REAL`, `REAL*COMPLEX`, and `COMPLEX/REAL`.

Some operations, such as absolute value and square root, are defined as functions, rather than operators.

There are functions that return the complex square root of a (possibly negative) real argument, and functions are provided for squaring complex numbers, as this can be done more efficiently than by simply multiplying them by themselves, which is significant in the power operator (§ 2.1).

The following constants are defined:

```
VAL COMPLEX32 Zero.COMPLEX32 IS [0.0, 0.0]:
VAL COMPLEX32  One.COMPLEX32 IS [1.0, 0.0]:
VAL COMPLEX32    I.COMPLEX32 IS [0.0, 1.0]:

VAL COMPLEX64 Zero.COMPLEX64 IS [0.0, 0.0]:
VAL COMPLEX64  One.COMPLEX64 IS [1.0, 0.0]:
VAL COMPLEX64    I.COMPLEX64 IS [0.0, 1.0]:
```

The obvious methods for some operations (division, absolute value, and square root) are liable to overflow and cancellation. The algorithms used attempt to avoid these errors.

2.5. Vectors

Vectors of the form $x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$, as used in mechanics, can be defined similarly:

```
DATA TYPE VECTOR32 IS [3]REAL32:
```

All the obvious operators are defined. The vector (outer, 'cross') product is represented by '`*`', and the scalar (inner, 'dot') product by `TIMES` ('`.`' is not allowed as an operator).

Quaternions are very similar.

2.6. `INT128`

This type is defined as

```
DATA TYPE INT128 IS [4]INT:
```

All the standard occam operators are provided.

`INT128` constants cannot be written directly. Sufficiently small values can be converted from `INT`s or `INT64`s, and the functions `DECTOINT128` and `HEXTOINT128` can be used to convert from strings of digits.

The following predefined constants are included:

```
VAL INT128 MinusOne.INT128 IS [-1, -1, -1, -1]:
VAL INT128     Zero.INT128 IS [ 0,  0,  0,  0]:
VAL INT128      One.INT128 IS [ 1,  0,  0,  0]:
VAL INT128      Two.INT128 IS [ 2,  0,  0,  0]:
VAL INT128      Ten.INT128 IS [10,  0,  0,  0]:
VAL INT128  MostNeg.INT128 IS [ 0,  0,  0,  MOSTNEG INT]:
VAL INT128  MostPos.INT128 IS [-1, -1, -1,  MOSTPOS INT]:
```

`Ten.INT128` is useful internally for conversions to and from decimal.

2.7. Multiple-Length Integers

These are a further extension of integer arithmetic to any predefined length.

The parameters of operators can be open arrays, but arrays returned as results must be of constant size. So the size of the numbers (in `INT`s) must be set before compiling the package; for example:

```
VAL INT LONG.Size IS 1000:
```

Then the type can be declared:

```
DATA TYPE LONG IS [LONG.Size]INT:
```

All the normal operators are defined. Multiplication and division are overloaded for `LONG*INT` and `LONG/INT`*.

The `LONG` factorial of an `INT` is represented by a prefix '`!!`' operator; for example, `!! 1000`.

As with `INT128`s, small constants can be converted from `INT`s of various sizes, and numbers of any size translated from strings of digits. `MOSTPOSLONG` and `MOSTNEGLONG` are provided as functions.

Scaled long integers can be used to implement multiple-length real numbers; for example, *e* to 1000 digits is easy.

This implementation is acceptable only for numbers of fairly moderate size, since time is wasted handling leading zeros. For serious work, a record including a length field should be used, as with `STRING`s (§ 2.16).

2.8. REAL128

These have been implemented for the SPARC and the MIPS, using system software provided on these machines. The formats are not the same: the SPARC is a natural extension of IEEE floating-point arithmetic [13]; the MIPS is not.

They are typically tens to hundreds of times slower than `REAL64`s.

The DEC Alpha also supports (software) `REAL128`s.

The type is declared as

```
DATA TYPE REAL128 IS [2]REAL64:
```

which should ensure the appropriate alignment. The two elements would not normally be accessed separately.

As with `INT128`, representing constants is difficult. Several are supplied, including, in SPARC format:

```
VAL REAL128 Zero.REAL128 RETYPES [0, 0, 0, 0]:
VAL REAL128 Half.REAL128 RETYPES [#3FFE0000, 0, 0, 0]:
VAL REAL128  One.REAL128 RETYPES [#3FFF0000, 0, 0, 0]:
VAL REAL128  Two.REAL128 RETYPES [#40000000, 0, 0, 0]:
VAL REAL128  Ten.REAL128 RETYPES [#40024000, 0, 0, 0]:

VAL REAL128   Pi.REAL128 RETYPES [#4000921F, #B54442D1,
                                  #8469898C, #C51701B8]:
VAL REAL128    E.REAL128 RETYPES [#40005BF0, #A8B14576,
                                  #95355FB8, #AC404E7A]:
```

* Actually, `LONG/LONG` is difficult, and not yet implemented.

In MIPS format, these are:

```
VAL REAL128 Zero.REAL128 RETYPES [0, 0, 0, 0]:
VAL REAL128 Half.REAL128 RETYPES [#3FE00000, 0, 0, 0]:
VAL REAL128  One.REAL128 RETYPES [#3FF00000, 0, 0, 0]:
VAL REAL128  Two.REAL128 RETYPES [#40000000, 0, 0, 0]:
VAL REAL128  Ten.REAL128 RETYPES [#40240000, 0, 0, 0]:

VAL REAL128   Pi.REAL128 RETYPES [#400921FB, #54442D18,
                                  #3CA1A626, #33145C07]:
VAL REAL128    E.REAL128 RETYPES [#4005BF0A, #8B145769,
                                  #3CA4D57E, #E2B1013A]:
```

Note that the following more natural form could be used only if `REAL64`s are in native format:

```
VAL REAL128 Zero.REAL128 IS [0.0, 0.0]:
VAL REAL128 Half.REAL128 IS [0.5, 0.0]:
VAL REAL128  One.REAL128 IS [1.0, 0.0]:
VAL REAL128  Two.REAL128 IS [2.0, 0.0]:
```

With these operations, it is easy to define `COMPLEX128`.

2.9. REAL80

A similar example allows the use of IEEE *double extended* precision, on those machines that support it, such as the 68881.

```
DATA TYPE REAL80 IS [3]INT:
```

This allocates 96 bits, as required for this type in memory by the 68881.

The Intel 80386 family uses a similar 80-bit floating-point format [14].

2.10. REAL16

The SHARC has a 16-bit floating-point format. This is declared as

```
DATA TYPE REAL16
  RECORD
    INT16 real16:
:
```

Simply using `INT16` would allow `REAL16` to inherit many inappropriate operators, such as `PLUS`.

The basic arithmetic and comparison operators are implemented.

2.11. Unsigned Integers

Checked arithmetic operators can be defined on unsigned integers of all sizes.

The unchecked arithmetic operators (`PLUS`, `MINUS`, and `TIMES`) are inherited from the signed types.

Signed `INT8`s could be provided similarly (`BYTE`s are already unsigned).

2.12. Rational Numbers

These are currently implemented as pairs of `INT64`s:

```
DATA TYPE RAT64
  RECORD
    INT64 num:                 -- signed
    INT64 den:                 -- always positive
:
```

To avoid overflow in intermediate values, they should perhaps use `INT128`s internally (§ 2.6), and the denominator should be unsigned (§ 2.11).

All the obvious operators are provided. Reciprocal is represented by '~'. This is a trivial operation for rational numbers, and is used internally to derive division from multiplication. The operator `X%Y` is used to create the rational value `X/Y` reduced to its lowest terms. It is overloaded for operands of several combinations of types. It is necessary because `[X, Y]` may not be a valid rational number; for example, `2%(-4)` is `[-1, 2]`.

Some constants are provided:

```
VAL RAT64      Zero.RAT64 IS [ 0, 1]:
VAL RAT64       One.RAT64 IS [ 1, 1]:
VAL RAT64  MinusOne.RAT64 IS [-1, 1]:
VAL RAT64  Infinity.RAT64 IS [ 1, 0]: -- not used
```

Continued fractions are used to convert from `REAL64`s to rationals.

2.13. Fixed-Point Numbers

These have advantages over floating-point numbers in some numerical applications (for example, they are not subject to cancellation errors). The data type might be declared as

```
DATA TYPE FIX64 IS INT64:
```

The number of bits in the integer and fractional parts can be defined with

```
VAL INT FIX64.Scale IS 32:      -- scale shift
```

Most operations, such as addition and subtraction, are inherited from `INT64`.

2.14. Matrices

Operators on matrices can be defined, provided that all the matrices in any one program are the same size. In general, this is a severe restriction, but might be tolerable in some applications, such as the transformation matrices in a graphical system.

```
VAL INT MATRIX.Size IS 4:

DATA TYPE VECTOR IS [MATRIX.Size]REAL64:
DATA TYPE MATRIX IS [MATRIX.Size]VECTOR:
```

All the obvious operations are available, including all the meaningful overloadings of '`*`' on combinations of variables of type `REAL64`, `VECTOR`, and `MATRIX`. Transpose is represented by '`@`', and inverse by '~'.

2.15. Sets

A SET data type, as in Pascal [15], can be implemented as a packed array of bits. Set operations then reduce to Boolean operations on such arrays (which may be a useful type in their own right.)

Unfortunately, as with multiple-length integers, the size of a set must be fixed at compile time. A reasonable value is 256 elements, giving a set of BYTEs, represented by eight INTs (or 16 on a 16-bit machine):

```
VAL INT SET.Bits  IS 256:       -- 2^BitsPerByte
VAL INT SET.Words IS 8:         -- SET.Bits/BitsPerWord

DATA TYPE SET IS [SET.Words]INT:
```

All the usual set operations are provided:

intersection	`SET * SET`	`SET /\ SET`
union	`SET + SET`	`SET \/ SET`
difference	`SET - SET`	
symmetric difference	`SET / SET`	`SET >< SET`
contained by	`SET <= SET`	
inclusion	`BYTE <= SET`	
contains	`SET >= SET`	
includes	`SET >= BYTE`	
(strict) superset	`SET > SET`	
(strict) subset	`SET < SET`	
inverse		`~ SET`

Overloading allows many useful combinations of sets and elements.

The operator '$$' is overloaded to make a set from a single byte, an open array of bytes, and a range of bytes.

A number of constant SETs are defined; for example: Empty.SET, UpperCase.SET, LowerCase.SET, Letter.SET, Digit.SET, Printing.SET.

2.16. Strings

Two approaches to implementing a string type have been investigated. In one, most of the work, including dynamic memory allocation, is done in C, and the occam data type is just an INT (or whatever is required to hold a pointer on the target machine). Problems with this are that the interface with occam arrays of BYTEs is complicated, particularly on big-endian machines, and assigning string variables introduces aliasing.

The second method works entirely in occam, and is therefore much cleaner. However, it suffers from the restriction that all strings in any one program must have the same maximum length. Here, the type definition is

```
DATA TYPE STRING
  RECORD
    INT length:
    [STRING.Max]BYTE string:
:
```

where length represents the current length of the string, and the constant STRING.Max must be defined before the package is compiled.

Alternatively, a fixed definition could be used:

```
DATA TYPE STRING
  RECORD
    BYTE length:
    [256]BYTE string:
:
```

Operators on strings include all the comparisons, and

```
STRING FUNCTION "+"  (VAL STRING S, T) ...
STRING FUNCTION "[>" (VAL STRING S, VAL INT N) ...
STRING FUNCTION "<]" (VAL STRING S, VAL INT N) ...
STRING FUNCTION "**" (VAL STRING S, VAL INT N) ...
INT    FUNCTION "/"  (VAL STRING S, T) ...
STRING FUNCTION "\"  (VAL STRING S, VAL RANGE R) ...
```

These represent, respectively, concatenation, head, tail, repetition, searching for one string in another (returning the index of the start of the substring if found, or −1 otherwise), and substring. `RANGE` is a special type:

```
DATA TYPE RANGE
  RECORD
    INT start, length:
:
```

which, in effect, allows the substring operator to have three operands.

The operator '`$$`' is overloaded to generate a `STRING` from an array of `BYTE`s or a single `BYTE`, and to copy a `STRING` efficiently, ignoring the unused elements.

Most of the operators can be overloaded for arrays of `BYTE`s, and single `BYTE`s, instead of `STRING`s.

The constant `Null.STRING` is defined.

2.17. Other Examples

It could be argued that `TIME`s, as returned by occam `TIMER`s, should constitute a type distinct from `INT`s. Sums or differences of `TIME`s are themselves `TIME`s, as are the products or quotients of `TIME`s and `INT`s. However, the quotient of two `TIME`s is an `INT`, and the product is an error. This would be a step in the direction of treating types as physical dimensions, which is a difficult subject.

On big-endian machines, new types could be defined for 64-bit integers and reals in native format. If the operators were translated in line (§ 3.2.9), this might be more efficient than reformatting.

The comparison operators can be defined for open arrays of any type.

An 'equivalence' operator is occasionally useful. It is defined for all integer data types; for example

```
INT INLINE FUNCTION "==" (VAL INT X, Y) IS ~(X >< Y):
```

A proper ‘modulo’ operator, ‘`%`’, could be defined, that behaved sensibly for negative operands; so, for example,

```
index := (index - 1) % size
```

would have the obviously desired effect. (There should also be a corresponding division operator. Ada has two operators, **rem** and **mod**, but only **rem** is consistent with the single division operator, ‘`/`’.)

Other data types supported by various machines could be accessed from occam; for example, IEEE *single extended* precision on some targets, and `REAL40` and saturating arithmetic on the SHARC.

Several modern machines have ‘SIMD’ instructions, such as the Pentium MMX [16] and SPARC VIS. Many of these operators would simply overload the standard ones. Software implementations would provide portability.

3. Implementation

The Kent Retargetable occam Compiler, KRoC, uses a modified version of the INMOS / SGS-THOMSON occam 2.1 *Toolset* compiler, `occ21`, to generate transputer assembly language output [17]. This is translated by a target-dependent translator, `octran`, into the assembly language of the target machine. The resulting code is assembled, and linked with a kernel and run-time libraries.

Implementing new operators involves additions to `occ21` to handle the syntax, and the provision of user-defined libraries to implement the operators.

3.1. Compiler Modifications

The front end of the compiler has been modified to convert operator definitions [18], such as

```
INT FUNCTION "%" (VAL INT X, Y) ...
```

into normal `FUNCTION`s, such as

```
INT FUNCTION udo.PER.INT.INT (VAL INT X, Y) ...
```

Then expressions like `x%y` are converted to function calls like `udo.PER.INT.INT (x,y)`.

The names of these user-defined operator functions depend on the types of their operands, so the same operators can be overloaded for any combination of operator types. They do not, however, depend on the types of their results; so, for example, the same square-root operator acting on a real argument could not be used to return a real or complex result depending on the context.

These names are invisible to the user, except in error messages and warnings from the compiler.

3.2. Writing Libraries

The example libraries described above have been implemented in a variety of ways, considered below.

3.2.1. *Precompiled* occam

The simplest method is to write the functions in occam. This has been done for most of the SET and RAT64 operators, and some of those for complex arithmetic. For example

```
SET FUNCTION "+" (VAL SET S, T) -- union
  SET Result:
  VALOF
    SEQ i = 0 FOR SET.Words
      Result[i] := S[i] \/ T[i]
    RESULT Result
:
```

Such functions can be precompiled into libraries.

3.2.2. *INLINE* occam

For simple operators, INLINE functions can be used, either as self-contained expressions, for example

```
COMPLEX32 INLINE FUNCTION "+" (VAL COMPLEX32 x, y) IS
  [x[real] + y[real], x[imag] + y[imag]]:
```

or by using expressions involving precompiled operators:

```
RAT64 INLINE FUNCTION "/" (VAL RAT64 p, q) IS p * (~q):
```

Following the transputer instruction set, for most of the user-defined types described here only the ‘>’ and ‘=’ comparisons are defined in full, the others being derived from them; for example

```
BOOL INLINE FUNCTION "<>" (VAL INT128 X, Y) IS NOT (X = Y):
BOOL INLINE FUNCTION "<"  (VAL INT128 X, Y) IS     (Y > X):
BOOL INLINE FUNCTION "<=" (VAL INT128 X, Y) IS NOT (X > Y):
BOOL INLINE FUNCTION ">=" (VAL INT128 X, Y) IS NOT (Y > X):
```

3.2.3. *Standard* occam *Libraries*

Some operations may correspond to functions already available from standard libraries. For example, the functions ROTATELEFT and ROTATERIGHT are defined in the occam maths function library [19], though their effect is undefined if the shift count is out of range. Hence the following could be used:

```
INT INLINE FUNCTION "@>" (VAL INT X, N) IS
                                   ROTATERIGHT (X, N /\ 31):
INT INLINE FUNCTION "<@" (VAL INT X, N) IS
                                   ROTATELEFT  (X, N /\ 31):
```

There are no such library functions for other data types.

For integers, the ‘extended arithmetic’ operators (§ 2.3) are essentially the lmul and ldiv instructions of the transputer [20], and so can be defined portably using the LONGPROD and LONGDIV functions from the occam multiple-length arithmetic library.

The shift operators, << and >>, can be overloaded for floating-point operands using the functions SCALEB and DSCALEB from the floating-point function library.

Powers with real exponents are difficult to compute accurately. The occam elementary function libraries provide the following:

```
REAL32 INLINE FUNCTION "^" (VAL REAL32 X, Y) IS  POWER(X, Y):
REAL64 INLINE FUNCTION "^" (VAL REAL64 X, Y) IS DPOWER(X, Y):
```

Alternatively, the C pow function could be used (§ 3.2.6).

For complex numbers, the elementary transcendental functions (EXP, LOG, etc.) can easily be written similarly; for example

```
COMPLEX32 INLINE FUNCTION COMPLEX32EXP (COMPLEX32 Z) IS
  EXP(Z[real]) * [COS(Z[imag]), SIN(Z[imag])] (COMPLEX32):
```

3.2.4. *Modified* occam *Libraries*

Sometimes a standard library can be modified to support a new type. For example, the INT64 functions for 16-bit transputers were converted into a INT128 library for 32-bit machines by a few systematic edits, replacing 64 by 128, 32 by 64, and 16 by 32, as necessary. A few constants were also changed in obvious ways, and most of the RETYPES declarations were simply deleted.

A typical operator is

```
INT128 FUNCTION "+" (VAL INT128 X, Y)
  INT128 Result:
  VALOF
    INT Carry:
    SEQ
      Carry, Result[0] := LONGSUM (X[0], Y[0], 0)
      Carry, Result[1] := LONGSUM (X[1], Y[1], Carry)
      Carry, Result[2] := LONGSUM (X[2], Y[2], Carry)
      Result[3] := LONGADD (X[3], Y[3], Carry)
    RESULT Result
:
```

Most of these functions are easy to extend to arbitrary-length numbers, giving the LONG library. For example

```
LONG FUNCTION "+" (VAL LONG X, Y)
  LONG Result:
  VALOF
    VAL INT Top IS LONG.Size - 1:
    INT Carry:
    SEQ
      Carry := 0
      SEQ i = 0 FOR Top
        Carry, Result[i] := LONGSUM (X[i], Y[i], Carry)
      Result[Top] := LONGADD (X[Top], Y[Top], Carry)
    RESULT Result
:
```

On some machines with special hardware it might be worth implementing these functions in assembly language. The 68000 has such hardware, though it is designed for big-endian representation.

Binary coded decimal might be a convenient alternative (again on the 68000).

A portable, though less efficient, REAL128 library could be produced in the same way.

3.2.5. *Transputer Assembly Language*

A few operators may be more conveniently written in transputer assembly language, using ASM sections in occam, but most are already available as library functions. Because of the way KRoC works, this would be portable.

3.2.6. C *and Target Libraries*

KRoC provides a simple method of calling C from occam [21]. This gives a convenient mechanism for accessing library functions on the target machine, as well as for writing functions that may be difficult to express in occam. The MIPS version of the REAL128 library was written in this way. The appropriate MIPS data type is represented by long double in C, and consists of two separately normalized REAL64s, so in this case the type defined above (§ 2.8) is an accurate description.

Operators are declared as external C functions; for example:

```
#PRAGMA EXTERNAL "PROC C.REAL128ADD *
                        * (REAL128 X, VAL REAL128 Y, Z) = 0"

REAL128 INLINE FUNCTION "+" (VAL REAL128 Y, Z)
  REAL128 X:
  VALOF
    C.REAL128ADD (X, Y, Z)
    RESULT X
:
```

C.REAL128ADD is the occam name of the C function _REAL128ADD:

```
void _REAL128ADD (int w[3])
{   *REAL128(w[0]) = VAL_REAL128(w[1]) + VAL_REAL128(w[2]);
}
```

REAL128 and VAL_REAL128 are additions to the standard KRoC package of macros for interfacing to C.

3.2.7. *Fortran*

Since the interface between C and Fortran is normally well defined (though not necessarily portable), the same method could be used to access library functions written in Fortran. Examples are COMPLEX32, which is COMPLEX in standard Fortran, and REAL128, which, as REAL*16, is a common extension.

3.2.8. *Target Assembly Language*

Other functions may not be easy to express in standard high-level languages. In this case, the assembly language of the target machine may be used, though this is, of course, not portable.

A 128-bit floating-point data type is defined in the SPARC architecture, with instructions such as `faddq`, though it is rarely, if ever, implemented in hardware. The recommended way of using it is through some library routines, called from assembly language.

Operators to use these routines can be defined as follows. First, an external function is declared; for example:

```
#PRAGMA EXTERNAL "REAL128 FUNCTION REAL128ADD *
                                   * (VAL REAL128 X, Y) = 0"
```

and a user-defined operator to call it:

```
REAL128 INLINE FUNCTION "+" (VAL REAL128 X, Y) IS
                                          REAL128ADD (X, Y):
```

Functions like `REAL128ADD` are implemented in SPARC assembly language as calls to the quadruple-precision library routines, like `__Q_add`:

```
        .global $REAL128ADD
$REAL128ADD:
        ld      [%l3+8],%o0      !  -> X
        ld      [%l3+12],%o1     !  -> Y
        ld      [%l3+4],%o2      !  -> result
        call    __Q_add          !  result := X + Y
        st      %o2,[%sp+64]     !* hidden parameter
        unimp   16               !  magic
        ld      [%l3],%o7        !  occam return
        retl
        inc     16,%l3           !*
!       ------------------------
```

`REAL128` elementary transcendental functions could be obtained from the Fortran library.

The IEEE *double extended* floating-point format is provided by some processors and co-processors in the 68000 family, and also by some Intel machines. For example

```
REAL80 INLINE FUNCTION "+" (VAL REAL80 X, Y) IS
                                           REAL80ADD (X, Y):
```

The function is implemented as

```
.REAL80ADD:
        fmove.x ([8,a6]),fp0    ! X
        fadd.x  ([12,a6]),fp0   ! Y
        fmove.x fp0,([4,a6])    ! $formalresult0
!       movea.l (a6),a0         ! occam return
        lea     16(a6),a6
        jmp     (a0)
!       ------------------------
```

On most machines, rotations are most easily implemented in assembly language.

The functions are first declared as external; then the operators are defined as calls to these functions:

```
INT INLINE FUNCTION "<@" (VAL INT X, N) IS
                                        INT32LROTATE (N, X):
INT INLINE FUNCTION "@>" (VAL INT X, N) IS
                                        INT32RROTATE (N, X):
```

The reversal of the operands is useful for optimization (§ 3.2.9).

The code is written in the target assembly language; for example, for the SPARC:

```
        .global $INT32LROTATE
$INT32LROTATE:
        ld      [%l3+4],%l2     !  N
        ld      [%l3+8],%l0     !  X
        sll     %l0,%l2,%l1     !  X << N  (modulo 32)
        neg     %l2,%l2
        srl     %l0,%l2,%l0     !  X >> (32 - N)
        or      %l0,%l1,%l0
!       ld      [%l3],%o7       !  occam return
        retl
        inc     16,%l3          !*
!       ------------------------
```

or for the 68000:

```
.INT32LROTATE:
        move.b  7(a6),d2        ! N
        move.l  8(a6),d3        ! X
        and.b   #31,d2          ! ?
        rol.l   d2,d3           ! X <@ N
!       move.l  (a6),a0         ! occam return
        lea     16(a0),a0
        jmp     (a0)
!       ------------------------
```

Operators for unsigned arithmetic can usually be implemented by compiling the corresponding functions for signed numbers, and then editing the assembly language to change the check for overflow to a check for carry. For example, on the SPARC, unsigned addition is

```
        addcc   %l0,%l1,%l0
        tlu     17              !+ OVERFLOW
```

3.2.9. *In-Line Target Assembly Language*

All these techniques (except for `INLINE` occam) carry the overhead of function calls. For some of the applications considered here, this is undesirable. To avoid it, a modification has been made to `octran`, the program used by KRoC to translate from transputer to target assembly language. Essentially the same mechanism that is used to look up transputer assembly-language instructions is reused with a predefined table of function names. The calls are then replaced by in-line code. This also enables simple optimizations to be performed when one operand is a constant.

The following example shows what can be done with rotation operations on the 68000. The statement `y := x <@ 42` compiles to a call of an external function, which may be written in occam or assembly language, as above:

```
        ldc     42
        call    L0              -- Call INT32LROTATE
```

On the 68000, this normally translates to

```
        moveq   #42,d2
        lea     LX7(pc),a0
        subq.w  #4,a6
        move.l  a0,d0
        movem.l d0/d2/d3,-(a6)
        bra     .INT32LROTATE
LX7:
```

With the in-lining option, this reduces to

```
        moveq   #42,d2
        and.b   #31,d2
        ror.l   d2,d3
```

Here `INT32LROTATE` is treated as a dummy function name; in effect, it represents an extension to the transputer instruction set, and `octran` translates it accordingly.

With constant optimization, this produces the optimal code

```
        swap    d3
        ror.l   #6,d3
```

Powers are most commonly used with small constant exponents. Unrolling the square-and-multiply loop into a sequence of multiplication instructions is a significant optimization. For example, on the SPARC, `x^10` generates

```
        fmuld   %f0,%f0,%f0
        fmuld   %f0,%f0,%f6
        fmuld   %f6,%f6,%f6
        fmuld   %f0,%f6,%f0
```

The SHARC supports its `REAL16` data type with two instructions, `fpack` and `funpack`, for converting between 16- and 32-bit formats. Hence most operations on this type are trivial, though fewer steps are needed in the iterations for division and square root. For example:

```
        f12 = funpack r9;           !- Call REAL16DIV
        f4 = funpack r5, f8 = m1;   !+ 2.0
        f0 = recips f4;             !+ approx. 1/den
        f12 = f0 * f12;             !+ refine
        f4 = f0 * f4, f0 = f8 - f12;
        f6 = f0 * f4;               !+ num * 1/den
        r5 = fpack f6;
```

4. Application to Standard Operators

Some built-in operators on the standard data types, particularly `INT16` and `INT64` for which the transputer hardware has limited support, are implemented as calls to ‘compiler library functions’. These are written in occam, and must be linked with any program that uses the operators concerned.

Such functions can, of course, be rewritten in the assembly language of the target machine, but they can also be treated exactly as described above. This is particularly advantageous for `INT16`s on machines that support 16-bit arithmetic in hardware, such as the 68000, and for `INT64`s on 64-bit machines, like the MIPS and the DEC Alpha.

For example, on the 68000, `INT16` division is:

```
ext.l   d2                  !- Call INT16DIV%CHK
divs.w  d3,d2
trapv                       !+ OVERFLOW
move.w  d2,d3
```

and on the Alpha, `INT64` multiplication is:

```
ldq     $2,($2)             #- Call INT64MUL%CHK
ldq     $1,($1)
mulqv   $2,$1,$2
stq     $2,($3)
```

Other standard operators may compile into long sequences of transputer instructions, which are quite inappropriate on machines that could do them directly. For example, on 64-bit machines, even 32-bit operations may sometimes incur an extra cost, so implementing 64-bit arithmetic as a sequence of 32-bit operations built up from 64-bit instructions is embarrassingly inefficient.

A solution to this is to redefine such operations as user-defined operators, using dummy functions that are then expanded in line by `octran` (taking care to preserve their semantics!). The resulting code can be close to optimal; on the MIPS, `INT64` division is more than four times faster.

For `INT64`s, the endianism of the target is important, and reformatting on a big-endian machine may neutralize the advantage of using native 64-bit arithmetic for some operations, so on the MIPS only multiplication and division are handled in this way, but on the DEC Alpha, which is little-endian, the advantage is considerable. A modification to the compiler ensures 64-bit alignment.

`INT64`s are passed to and from functions by reference, so the resulting code accesses such variables indirectly, introducing an additional overhead. But the transputer instruction set requires this anyway for floating-point operations, so the structure of the translated code for `INT64`s and `REAL64`s can be quite comparable. For complex expressions there are some redundant stores and loads, but even a simple peephole optimizer could remove most of these.

Using these techniques, `INT16`s are handled entirely in line on the 68000, using native 16-bit arithmetic, so no library is needed to support them. On the MIPS and DEC Alpha, the `INT64` library is eliminated as well.

Some floating-point operators could be treated similarly; for example, the transputer has no floating-point ‘negate’ instruction, so `-x` is compiled as `0.0 - x`. Unfortunately, the most difficult cases, such as conversions between `REAL64`s and `INT64`s, currently have to be written as special functions. It is intended to modified the compiler solve this problem.

5. Other Applications

It may be convenient to redefine some common functions, such as absolute value and square root, as unary operators. This enables the same source code to be used to implement operations on a range of similar types, only the declarations needing to be changed. For example, most of the power functions could be written in this way.

Absolute value can be written as follows for `INT`:

```
INT INLINE FUNCTION "@@" (VAL INT X)
  INT Result:
  VALOF
    IF
      X < 0
        Result := -X
      TRUE
        Result :=  X
    RESULT Result
:
```

and similarly for all integer data types. For other types, it is either a standard function or has been defined in the relevant library:

```
REAL16     INLINE FUNCTION "@@" (VAL REAL16     X) IS
                                               REAL16ABS (X):
REAL32     INLINE FUNCTION "@@" (VAL REAL32     X) IS
                                                     ABS (X):
REAL64     INLINE FUNCTION "@@" (VAL REAL64     X) IS
                                                    DABS (X):
REAL128    INLINE FUNCTION "@@" (VAL REAL128    X) IS
                                              REAL128ABS (X):
COMPLEX32  INLINE FUNCTION "@@" (VAL COMPLEX32  X) IS
                                            COMPLEX32ABS (X):
COMPLEX64  INLINE FUNCTION "@@" (VAL COMPLEX64  X) IS
                                            COMPLEX64ABS (X):
COMPLEX128 INLINE FUNCTION "@@" (VAL COMPLEX128 X) IS
                                           COMPLEX128ABS (X):
```

Square root is similarly available for all relevant data types; for example

```
REAL32     INLINE FUNCTION "%%" (VAL REAL32 X) IS  SQRT (X):
REAL64     INLINE FUNCTION "%%" (VAL REAL64 X) IS DSQRT (X):
```

Since overloaded operators are identified by the types for their parameters, not of their results, the same operator could not be used for the complex square roots of (possibly negative) real arguments.

6. Separate Compilation

The sizes of some types (multiple-length integers, sets, strings, etc.) must be defined before the functions implementing operations on them can be compiled, so libraries of such operations must be `#INCLUDE`d as text, not `#USE`d as precompiled code.

This problem can be largely avoided by writing the algorithms as `PROC`s, using open array parameters, which can be compiled separately. These can then be called by small (possibly `INLINE`) interface `FUNCTION`s, using `RETYPES` to pass the actual parameters, results, and possibly temporary workspace to the precompiled `PROC`s. However, this generates less efficient code.

7. Names

A systematic naming convention has been used here, based on that for the compiler library functions. It may be useful to translate these names, to avoid any possibility of clashes with user-defined functions. For example:

```
#PRAGMA TRANSLATE INT32LROTATE "INT32LROTATE%UDO"
```

This has been done for the functions translated in line by `octran`.

Types names are all upper case.

Constant instances of a type have names of the form *`Name.TYPE`*, and parameters defining a type have names of the form *`TYPE.Name`*.

8. Usage

New operators can be obtained using `#INCLUDE` files; for example:

```
#INCLUDE "real128.inc"
```

This in turn consists of something like:

```
#INCLUDE "real128.def"          -- type definition
#USE     "real128.tco"          -- from compiled code
#INCLUDE "real128.inl"          -- INLINE definitions
```

The library will have been compiled from `real128.occ`, which will itself `#INCLUDE real128.def`, and possibly `real128.inl` (the dependencies between compiled and `INLINE` functions can be complicated):

```
kroc -c real128.occ
```

This will produce the files `real128.tco` and `real128.o`.

A user program is then compiled with a command of the form

```
kroc prog.occ real128.a
```

where `real128.a` is a library containing `real128.o` and routines written in `C` or assembly language, `r128.o`.

There is an implementation restriction in the compiler that ‘Expression for outermost level VAL must be constant’. Since some constants of user-defined types may be defined in terms of their operators, for example

```
VAL STRING Null.STRING IS $$ "":
```

these packages should be `INCLUDE`d inside, not outside, the main `PROC`.

9. Discussion

User-defined operators provide a useful addition to occam. They are semantically well behaved, in that they can have no side effects, because they are just a ‘syntactic sugaring’ of FUNCTIONs. So expressions retain their referential transparency − nothing has been done that damages occam security.

As pointed out above, when used as the results of FUNCTIONs (and hence of operators), types involving arrays must be of a fixed size. This is a severe restriction that limits the use of several of the data types described, and makes a few, such as matrices, almost useless.

We have been considering more significant changes to occam syntax to solve this problem [22]. Consider the case of matrix multiplication. It is easy to write a PROC that accepts open arrays as its parameters:

```
PROC MATMULT (VAL [][]REAL64 X, Y, [][]REAL64 Z)
  VAL INT P IS SIZE X:
  VAL INT Q IS SIZE Y[0]:
  VAL INT R IS SIZE X[0]:
  SEQ
    ASSERT ((SIZE Z)    = P)    -- check consistency
    ASSERT ((SIZE Z[0]) = Q)
    ASSERT ((SIZE Y)    = R)
    ...
:
```

The sizes of the arrays are already passed as hidden extra parameters. Here we have simply named them by the abbreviation mechanism.

The proposed syntax make these sizes explicitly visible, as parts of the parameters, so that checks can be made automatically by the compiler, either statically or dynamically:

```
PROC MATMULT (VAL [VAL INT P][VAL INT Q]REAL64 X,
              VAL [        Q][VAL INT R]REAL64 Y,
                  [        P][        R]REAL64 Z)
  ...
:
```

Some of the dimensions are no longer open, as they depend on those of earlier parameters.

Structured function results are currently passed by reference, but arrays must be of a fixed size. The proposed syntax relaxes this restriction, and requires passing the size as well. The result becomes an explicit parameter; it is placed at the end to make the dependencies more natural:

```
FUNCTION "**" (VAL [VAL INT P][VAL INT Q]REAL64 X,
               VAL [        Q][VAL INT R]REAL64 Y) ->
                   [        P][        R]REAL64 Z
  ...
:
```

References

[1] INMOS Limited. occam 2 Reference Manual; Prentice Hall, 1988. ISBN 0−13−629312−3.

[2] David C. Wood and Peter H. Welch. The Kent Retargetable occam Compiler; *Proceedings of WoTUG-19: Parallel Processing Developments*; IOS Press, 1996. ISBN 90−5199−261−0.

[3] Van Wijngaarden et al. Revised Report on the Algorithmic Language ALGOL 68. 1974.

[4] Reference Manual for the Ada Programming Language. Ada Joint Program Office. 1982.

[5] ISO/IEC. Information Technology − Programming Languages − Fortran (ISO/IEC 1539:1991(E)). ISO/IEC Copyright Office, Geneva, 1991.

[6] SPARC International. The SPARC Architecture Manual. Prentice Hall, 1992. ISBN 0−13−825001−4.

[7] Richard L. Sites. Alpha Architecture Reference Manual. Digital Press, 1992. ISBN 1−55558−098−X / 0−13−033663−7.

[8] Charles Price. MIPS IV Instruction Set. MIPS Technologies, Inc., 1995.

[9] Motorola. M68000 8- / 16- / 32-Bit Microprocessors Programmer's Reference Manual, Fifth edition. Prentice-Hall, 1986. ISBN 0−13−541475−X.

[10] Analog Devices. ADSP-2106x SHARC User's Manual, Second Edition.

[11] Conor O'Neill. occam-2.5 definition. 1994.

[12] ANSI. Programming Language FORTRAN. X3.9-1978. ANSI, New York, 1978.

[13] ANSI / IEEE Std 754-1985. IEEE Standard for Binary Floating-Point Arithmetic. The Institute of Electrical and Electronic Engineers, Inc, 1985.

[14] John H. Crawford and Patrick P. Gelsinger. Programming the 80386. SYBEX, 1987. ISBN 0−89588−381−3.

[15] K. Jensen and N. Wirth. Pascal User Manual and Report. Springer-Verlag, 1975.

[16] Alex Peleg, Sam Wilkie, and Uri Weiser. Intel MMX for Multimedia PCs. *Communications of the ACM*, January 1997 / Vol. 40, No 1.

[17] David Wood. KRoC − An Implementors' Guide. University of Kent at Canterbury. 1998.

[18] James Moores. User-Defined Operators in occam 2.x. University of Kent at Canterbury. 1998.

[19] INMOS. occam 2 Language Toolset Language and Libraries Reference Manual. INMOS Limited, 1993.

[20] INMOS Limited. The transputer instruction set − a compiler writers' guide. Prentice Hall, 1988. ISBN 0−13−929100−8.

[21] David Wood. KRoC − Calling C Functions from occam. University of Kent at Canterbury. 1998.

[22] Peter Welch. Private communication. 1999.

Appendices

A. Operators

The following table lists most of the operators considered in this paper. Some have been implemented on several target machines; others only on those that support the data types concerned.

Arithmetic	+	−	*	/	\	^	@	~	!!
BYTE	+	+	+	+	+	*			
INT16	+	+	+	+	+	*	*		
INT	+	+	+	+	+	*	*		
INT64	+	+	+	+	+	*	*		*
INT128	*	*	*	*	*	*			*
LONG	*	*	*	*	*	*			*
UNSIGNED	*	*	*	*	*		+		
REAL16	*	*	*	*	*		*		
REAL32	+	+	+	+	+	*	*		
REAL64	+	+	+	+	+	*	*		
REAL128	*	*	*	*	*	*			
COMPLEX32	*	*	*	*		*		*	
COMPLEX64	*	*	*	*		*		*	
COMPLEX128	*	*	*	*		*		*	
RAT64	*	*	*	*				*	
VECTOR	*	*	*						
MATRIX	*	*	*	*			*	*	
SET	*	*	*	*				*	
STRING	*		*	*	*	*	*	*	
Comparisons	=	<>	>	>=	<	<=			
INT128	*	*	*	*	*	*			
LONG	*	*	*	*	*	*			
UNSIGNED	+	+	*	*	*	*			
REAL16	*	*	*	*	*	*			
REAL32	+	+	+	+	+	+			
REAL64	+	+	+	+	+	+			
REAL128	*	*	*	*	*	*			
COMPLEX32	*	*							
COMPLEX64	*	*							
COMPLEX128	*	*							
RAT64	*	*	*	*	*	*			
VECTOR	*	*							
MATRIX	*	*							
SET	*	*	*	*	*	*			
STRING	*	*	*	*	*	*			
Logic	/\	\/	><	==	~	<<	>>	<@	@>
BYTE	+	+	+	*	+	+	+	*	*
INT16	+	+	+	*	+	+	+	*	*
INT	+	+	+	*	+	+	+	*	*
INT64	+	+	+	*	+	+	+	*	*
INT128	*	*	*	*	*	*	*	*	*
LONG	*	*	*	*	*	*	*	*	*
UNSIGNED	+	+	+	*	+	+	+	+	+
Unchecked	PLUS	MINUS	TIMES	AFTER					
INT128	*	*	*	*					
LONG	*	*	*	*					
UNSIGNED	+	+	+	+					

Key:

+	Predefined in occam or inherited
*	New operator, sometimes overloaded for several combinations of operand types

B. Functions

Many useful operations are provided as functions, rather than operators.

Result	*Function*	*Parameters*
COMPLEX32	COMPLEX32SQR	**(VAL COMPLEX32)**
COMPLEX32	COMPLEX32SQRT	**(VAL COMPLEX32)**
COMPLEX32	COMPLEX32SQRTREAL	**(VAL REAL32)**
REAL32	COMPLEX32ABS	**(VAL COMPLEX32)**
REAL32	COMPLEX32ABSSQ	**(VAL COMPLEX32)**
COMPLEX64	COMPLEX64SQR	**(VAL COMPLEX64)**
COMPLEX64	COMPLEX64SQRT	**(VAL COMPLEX64)**
COMPLEX64	COMPLEX64SQRTREAL	**(VAL REAL64)**
COMPLEX32	COMPLEX64ROUND	**(VAL COMPLEX64)**
COMPLEX32	COMPLEX64TRUNC	**(VAL COMPLEX64)**
REAL64	COMPLEX64ABS	**(VAL COMPLEX64)**
REAL64	COMPLEX64ABSSQ	**(VAL COMPLEX64)**
COMPLEX128	COMPLEX128SQR	**(VAL COMPLEX128)**
COMPLEX128	COMPLEX128SQRT	**(VAL COMPLEX128)**
COMPLEX128	COMPLEX128SQRTREAL	**(VAL REAL128)**
COMPLEX64	COMPLEX128ROUND	**(VAL COMPLEX128)**
COMPLEX64	COMPLEX128TRUNC	**(VAL COMPLEX128)**
REAL128	COMPLEX128ABS	**(VAL COMPLEX128)**
REAL128	COMPLEX128ABSSQ	**(VAL COMPLEX128)**
REAL16	REAL16ABS	**(VAL REAL16)**
REAL16	REAL16SQRT	**(VAL REAL16)**
REAL128	REAL128ABS	**(VAL REAL128)**
REAL128	REAL128SQRT	**(VAL REAL128)**
REAL128	REAL128CEIL	**(VAL REAL128)**
REAL128	REAL128FLOOR	**(VAL REAL128)**
REAL128	REAL128ROUND	**(VAL REAL128)**
REAL128	REAL128TRUNC	**(VAL REAL128)**
REAL128	REAL128SCALE	**(VAL REAL128, VAL INT)**
REAL128	REAL128MANTISSA	**(VAL REAL128)**
INT	REAL128EXPONENT	**(VAL REAL128)**
REAL128, INT	REAL128MANEXP	**(VAL REAL128)**
INT	REAL128CEIL32	**(VAL REAL128)**
INT	REAL128FLOOR32	**(VAL REAL128)**
INT	REAL128ROUND32	**(VAL REAL128)**
INT	REAL128TRUNC32	**(VAL REAL128)**
INT64	REAL128CEIL64	**(VAL REAL128)**
INT64	REAL128FLOOR64	**(VAL REAL128)**
INT64	REAL128ROUND64	**(VAL REAL128)**
INT64	REAL128TRUNC64	**(VAL REAL128)**
INT128	REAL128CEIL128	**(VAL REAL128)**
INT128	REAL128FLOOR128	**(VAL REAL128)**
INT128	REAL128ROUND128	**(VAL REAL128)**
INT128	REAL128TRUNC128	**(VAL REAL128)**
INT128, INT128	INT128DIVREM	**(VAL INT128, VAL INT128)**
INT128, INT	INT128DIVREM32	**(VAL INT128, VAL INT)**
INT, INT128	INT128NORM	**(VAL INT128)**
LONG, INT	LONGDIVREM32	**(VAL LONG, VAL INT)**
INT, LONG	LONGNORM	**(VAL LONG)**
LONG	LONGMOSTPOS	**( )**
LONG	LONGMOSTNEG	**( )**
INT	SETMEMBERS	**(VAL SET)**

C. Type Conversion Functions

For *n* data types, $O(n^2)$ type-conversions are required. Not all have been written yet, hence some of the ellipses below.

Complex Numbers

From To:	**COMPLEX32**	**COMPLEX64**	**COMPLEX128**
REAL32	REAL32TOCOMPLEX32	. . .	. . .
REAL64	. . .	REAL64TOCOMPLEX64	. . .
REAL128	. . .	. . .	REAL128TOCOMPLEX128
COMPLEX32	–	COMPLEX32TOCOMPLEX64	COMPLEX32TOCOMPLEX128
COMPLEX64	COMPLEX64ROUND/TRUNC	–	COMPLEX64TOCOMPLEX128
COMPLEX128	COMPLEX128TOCOMPLEX32	COMPLEX128TOCOMPLEX64	–

Integer and Real Numbers

From To:	**INT**	**INT64**	**INT128**	**LONG**
INT	–	*INT64*	INT32TOINT128	INT32TOLONG
INT64	*INT ROUND/TRUNC*	–	INT64TOINT128	INT64TOLONG
INT128	INT128TOINT32	INT128TOINT64	–	INT128TOLONG
LONG	LONGTOINT32	LONGTOINT64	LONGTOINT128	–
REAL32	*INT ROUND/TRUNC*	*INT64 ROUND/TRUNC*	REAL32TOINT128	REAL32TOLONG
REAL64	*INT ROUND/TRUNC*	*INT64 ROUND/TRUNC*	REAL64TOINT128	REAL64TOLONG
REAL128	REAL128TOINT32	REAL128TOINT64	REAL128TOINT128	REAL128TOLONG
BYTE	*INT*	*INT64*	BYTETOINT128	BYTETOLONG
[]BYTE	. . .	. . .	DEC/HEXTOINT128	DEC/HEXTOLONG
From To:	**REAL16**	**REAL32**	**REAL64**	**REAL128**
INT	. . .	*REAL32 ROUND/TRUNC*	*REAL64 ROUND/TRUNC*	INT32TOREAL128
INT64	. . .	*REAL32 ROUND/TRUNC*	*REAL64 ROUND/TRUNC*	INT64TOREAL128
INT128	. . .	INT128TOREAL32	INT128TOREAL64	INT128TOREAL128
LONG	. . .	LONGTOREAL32	LONGTOREAL64	LONGTOREAL128
REAL16	–	REAL16TOREAL32	. . .	. . .
REAL32	REAL32TOREAL16	–	*REAL64*	REAL32TOREAL128
REAL64	. . .	*REAL32 ROUND/TRUNC*	–	REAL64TOREAL128
REAL128	. . .	REAL128TOREAL32	REAL128TOREAL64	–
BYTE	. . .	*REAL32 ROUND/TRUNC*	*REAL64 ROUND/TRUNC*	BYTETOREAL128
[]BYTE	. . .	. . .	. . .	DECTOREAL128

Rational Numbers

From To:	**REAL64**	**RAT64**
REAL64	–	REAL64TORAT64
RAT64	RAT64TOREAL64	–

Unsigned Integers

From To:	**UNSIGNED32**	**REAL32**	**REAL64**	**REAL128**
UNSIGNED32	–	UNS32TOREAL32	UNS32TOREAL64	UNS32TOREAL128
REAL32	REAL32TOUNS32	–	. . .	. . .
REAL64	REAL64TOUNS32	. . .	–	. . .
REAL128	REAL128TOUNS32	. . .	. . .	–

Output

From To:	*Decimal*	*Hexadecimal*
INT128	INT128TODEC	INT128TOHEX
LONG	LONGTODEC	LONGTOHEX
REAL128	REAL128TODEC	. . .