occam-2.5 definition

Conor O’Neill

1 March, 1994

1 Named types

1.1 Description

Permit the programmer to name basic scalar (and array) types, so that they can be referenced by more ‘abstract’ and/or memorable names.

Named types provide enhanced type security because each named type may not be mixed with any other inadvertently.

1.2 Rationale

This proposal is included directly from Geoff Barrett’s work on occam 3. See section 5.1 of Draft occam3 reference manual.

1.3 Syntax

Geoff provides the following syntax: (Note - Geoff’s document has changed type to data.type)

\[
definition \quad = \quad \text{DATA TYPE} \quad \text{name IS type :}
\]

\[
type \quad = \quad \text{name}
\]

Named array tables are defined in a similar way to tables, except that the name of the array type is written at the end as a type decoration:

\[
table \quad = \quad [ \{ 1 \ , \ \text{expression} \} \ ] \quad ( \text{type})
\]

New keywords: DATA, TYPE.

This has two real implications on parsing. Parsing the basic type declaration is trivial. However, now whenever a type may appear, we must permit a name also.

As far as I can tell, the affected situations (pre changes) are as follows:
1.4 Semantics

A named type introduces a new type with the same structure as the type listed on the right-hand side. These types, however, are not compatible, and may not be mixed without an explicit type conversion.

Two named data types are equal only when they refer to the same type declaration. For example, in:

```
DATA TYPE AREA IS REAL32 :
AREA a, b :
REAL32 c :
```

the types of a and b are the same, but the type of c is different. In:

```
DATA TYPE AREA IS REAL32 :
AREA a :
```
DATA TYPE AREA IS REAL32 :
AREA b :

the types of a and b are different because the two declarations introduce different named types, even though they are spelt the same.

Note - in this respect, named types behave identically to named protocols.
MOSTPOS and MOSTNEG may be applied to named types which are derived from an integer type.
(Also BYTE, see section 8.)
A named array data type is not type-equivalent to any other similarly sized named array data type. A consequence of this is that constructors for named array data types must have the name of the type appended as a type decoration, except where that type can be deduced, as described in section 6.

1.5 Constraints
- A type used in a named data type declaration must be a primitive data type, another named data type, or an array of such, where the size of the array must be compile-time constant.

1.6 Other points
Named types must be exportable from separately compiled modules, both as exported objects themselves, and in the parameter lists of exported procedures and functions. We implement this in the same way as we currently do for protocols – i.e. we require the use of include-files.

2 Record types

2.1 Description
A way to collect together related data and to refer to the collection by means of a single name.

2.2 Rationale
This proposal is included directly from Geoff Barrett’s work on occam 3. See section 5.2 of Draft occam3 reference manual.

2.3 Syntax

definition = DATA TYPE name
             structured.type

structured.type = RECORD
{ declaration }

New keywords: RECORD.
Record literals (tables) are defined in a similar way to array tables, except that the name of the record type is written at the end as a type decoration:
table = [ { 1 , expression } ] ( type )

Note that record fields are accessed with the same syntax as array subscripts. The field name replaces the array index.
element = element [ subscript ]

subscription = expression
expression = operand
operand = element
element = name
2.4 Semantics

The declarations inside a *structured type* must be simple data declarations. No channels, timers, ports, procedures, functions, protocols, etc, may be declared there.

Records may contain fields which are themselves records, but only by creating a named type for the sub-record. Records may contain fields which are arrays, and arrays of records may be constructed.

The language does not specify anything about how the fields are actually ordered and/or packed into memory. An implementation may re-order fields, and insert padding between fields and at the end of the record, to facilitate access to arrays of records. (See section 3 for methods of controlling the layout of record structures in memory.)

Record literals, however, must have fields specified in the same order as the fields are declared.

The same field name may be used in the definition of more than one record type. The correct record type is deduced from context. It is an error if the same field name is used more than once in a single record type. See also section 12 with regard to scoping of protocol tags.

Zero length records are permitted. It is not particularly clear whether they are useful.

A record may be used in more than one component of a parallel, provided that different fields are used.

2.5 Other points

A multiprocessor implementation may restrict the types of records (and *packed* records, see section 3) which may be communicated between different processors. Typically, this will be because of word size, alignment, and endian-ness issues.

Thus, for example, an implementation might restrict the use of records containing *int* fields, or those which naturally align to different boundaries on different word lengths.

3 Packed Record types

3.1 Description

Adds a mechanism for controlling the concrete representation of *record* data types.

3.2 Rationale

This permits a programmer to use *record* data types to map directly onto externally specified data formats, such as those required by external hardware, or communications protocols, etc.

Geoff proposes different mechanisms for this in Appendix D of Draft *occam3* reference manual. However I find his syntax obscure, and the flexibility of *width* overly complex.

3.3 Syntax

```plaintext
structured.type = PACKED RECORD

{ declaration }

New keywords: PACKED.
```

3.4 Semantics

A record type may be declared as a *packed record*. This type is then different from any similarly shaped *record*.

The fields of a *packed record* will be laid out in memory in the order specified in the declaration of the record. The implementation may not re-order fields, nor may it introduce padding between fields, or at the end.
The layout of a nested field of RECORD type is not specified unless that type is PACKED. Otherwise it is unspecified which order the bytes of a multiple-byte field are stored in memory.

An implementation may restrict the PACKED RECORDs which it accepts; an implementation may insist that data types are only placed at appropriate boundaries, or that BYTESIN a PACKED RECORD be a multiple of the machine's word size. However, any record declaration which is accepted must be treated as a first class object; it may be assigned, and passed as a parameter, etc.

Note that it is vital that it is made clear in the documentation and/or diagnostics that there might be implementation restrictions here.

3.5 Other points

Note that 'direct' type conversion between PACKED and non-packed RECORDs is not supported. A user may supply explicit conversion functions, which perform field-by-field copying. See section 9.

An implementation must define the size and alignment requirements of each of the primitive types, together with user defined types, record types, and arrays.

4 BYTESIN operator

4.1 Description

Permit access to the size of a data type (and/or variable), in bytes. We also extend the semantics and syntax of the SIZE operator to permit it to be applied to user defined types.

4.2 Rationale

This is needed to find out the size of a record type. It is obvious that if implemented, it should not be restricted just to record types.

I've just discovered that Geoff had already proposed this as WIDTHOF. See section D.5 of Draft occam3 reference manual. I find WIDTHOF to be counter-intuitive, and others have agreed that they prefer BYTESIN.

See also OFFSETOF in section 5.

Given that BYTESIN should work on types, it seems obvious that SIZE should too.

4.3 Syntax

By analogy with the SIZE operator, BYTESIN should be simply a monadic operator. However, this won't permit operating directly on a type rather than on a item of a type.

expression = monadic.operator operand

Presumably this implies (though it is not specified):

monadic.operator = SIZE | + | - | ...

To permit SIZE to operate on types, we must add:

expression = SIZE type
operand = BYTESIN ( operand )
| BYTESIN ( type )

Note the presence of parentheses around the operand or type. This makes it look like a function call, and is why it is defined as an operand, rather than an expression.

New keywords: BYTESIN.

4.4 Semantics

BYTESIN returns an INT containing the number of bytes required to represent the type/object.

Consider

x := BYTESIN ( expression ),
For objects and expressions, this is equivalent to:

```
VAL □BYTE temp RETYPES expression :
  x := SIZE temp
```

The value returned by \texttt{BYTESIN} for a type will be the number of bytes which are required when the type is used in an array. Thus in a record type, it would include any trailing padding. This also implies that, for all types:

```
BYTESIN (\{n\} type) == n * ( BYTESIN (type) )
```

It is invalid to apply the \texttt{BYTESIN} operator to an object whose size cannot be represented as an integral number of bytes. This might occur because it is too large, or because the object is not packed into an integral number of bytes, eg. \texttt{BOOL} objects might be packed into single bits. \textit{(Note - this is no different than the current situation with RETYPES.)}

\texttt{SIZE} can now be applied to a type, including user defined types. It is invalid to apply it to any type which isn’t an array. \texttt{SIZE} returns the number of elements in the array.

\texttt{BYTESIN} is syntactically an \textit{operand}, (unlike \texttt{SIZE} which is an \textit{expression}), so that it looks syntactically like a function call, and doesn’t need superfluous brackets when used in expressions. Hence it may be used in circumstances where \texttt{SIZE} is not permitted, eg:

```
x := BYTESIN(a) + BYTESIN(b) -- legal
x := SIZE a + SIZE b      -- illegal
x := (SIZE a) + (SIZE b)  -- legal
```

5 \textbf{OFFSETOF} operator

5.1 Description

An operator which returns the ‘offset’ in bytes of any particular field of a record or packed record type.

5.2 Rationale

Allow low-level manipulation of record types, to permit the use of \texttt{RETPES} etc. See also \texttt{BYTESIN} in section 4.

5.3 Syntax

```
operand = OFFSETOF ( name , name )
```

New keywords: \texttt{OFFSETOF}.

5.4 Semantics

\texttt{OFFSETOF ( type , name )} returns an \texttt{INT} containing the number of bytes (as determined by the implementation’s layout of the record) from the start of an object of record type \textit{type} to the beginning of field \textit{name}.

It is invalid if \textit{type} is not the name of a record or packed record type, or if \textit{name} is not a field of that record type.

It is invalid to apply the \texttt{OFFSETOF} operator to a field whose offset cannot be represented as an integral number of bytes. This might occur because it is too large, or because the field is not aligned to an integral number of bytes, eg. \texttt{BOOL} objects might be packed into single bits.

\texttt{OFFSETOF} is syntactically an \textit{operand}, so that it looks syntactically like a function call, and doesn’t need superfluous brackets when used in expressions.

```
x := OFFSETOF(type1, field1) + OFFSETOF(type2, field2) -- legal
```
6 Typed literals

6.1 Description
Permits scalar and array literals to have their type deduced from the context in which they are written, so that a user does not need to add the type decorations to literals.

6.2 Rationale
A common complaint among occam 2 programmers is that literals (especially real-valued literals) are cumbersome to write because of the need to follow each literal value with a type specifier.

In most cases the compiler could deduce this itself.

6.3 Syntax
The only syntactic change is to permit untyped real literals:

```
literal = real
```

6.4 Semantics
A type decoration may be omitted when there is only one decoration which would type-check correctly. In the following circumstances contextual information is used to deduce type:

- Inside a single expression.
  In this context, no information about expression types is passed either ‘into’ or ‘out of’ a VALOF expression.

- Expressions in process constructs where only one data type is permitted are assumed to have that type:
  - Array size and subscript expressions must be of type INT.
  - Start and length expressions of segments must be of type INT.
  - Guards of conditional processes and loops, and boolean guards of alternatives, must be of type BOOL.
  - Start and length expressions of replicator variables must be of type INT.
  - Shift counts must be of type INT.

- Assignment and output use the types of the variables or protocol of the channel.

- In abbreviations, the type of the expression is inferred from the type of the abbreviation. This rule also applies to the actual parameters of functions and procedures.

- The types of expressions in a RESULT list of a function use the return type of the function.

- The types of constant expressions in a CASE selection use the type of the selecting CASE expression.

- Expressions inside a table construct use the type of the known context of the table, together with any optional literal decoration, to deduce the types of each expression.

7 BYTE arithmetic

7.1 Description
Permit the integer arithmetic operations on BYTE variables.
7.2 Rationale

It is cumbersome to convert BYTEs to INTs and back again every time it is required to do any arithmetic or bit manipulation. These conversions may introduce many conversion checks.

7.3 Syntax

No extensions required.

7.4 Semantics

Arithmetic operations are permitted on BYTEs as follows. Overflow occurs (where applicable) when the value of the result exceeds the range 0 to 255 inclusive.

<table>
<thead>
<tr>
<th>Monadic operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>MINUS</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Conversion operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROUND</td>
</tr>
<tr>
<td>TRUNC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dyadic operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
</tr>
<tr>
<td>-</td>
</tr>
<tr>
<td>*</td>
</tr>
<tr>
<td>/</td>
</tr>
<tr>
<td>/\</td>
</tr>
<tr>
<td>&gt;</td>
</tr>
<tr>
<td>&gt;&gt;</td>
</tr>
</tbody>
</table>

Note that AFTER needs some further description. 'x AFTER y' is specified as ‘(x MINUS y) > 0’ for all current data types. However, with an unsigned data type, the comparison against zero succeeds for all non-zero values, making 'x AFTER y' equivalent to ‘x <> y’. Instead, (for BYTEs only), we must effectively consider the result of the MINUS operator to be a signed type before comparing against zero, giving

\[
x \text{ AFTER } y = ((x \text{ MINUS } y) > 0) \text{ AND } ((x \text{ MINUS } y) < 128)
\]

Type conversions are permitted on BYTE objects in the same way as other integer types. Thus, conversion to and from REAL32 and REAL64 are permitted, by using the ROUND and TRUNC operators. Note that conversions between BYTE and the other integer types are already permitted in occam 2.

8 MOSTPOS and MOSTNEG BYTE

8.1 Description

Extend MOSTPOS and MOSTNEG to BYTE types, and those types derived from BYTE.

8.2 Rationale

This means greater abstraction; it is possible to use MOSTPOS and MOSTNEG on named types without caring whether they are implemented as an integer type or as BYTE.
8.3 Syntax
No extensions required.

8.4 Semantics
MOSTPOS BYTE evaluates to 255(BYTE). MOSTNEG BYTE evaluates to 0(BYTE).

9 FUNCTIONs returning fixed-length objects

9.1 Description
FUNCTIONs should be permitted to return records, and fixed length arrays.

9.2 Rationale
It is obvious that FUNCTIONs must be able to return user-defined types to be usable. Given this, they should be able to return record types, and arrays whose size are known at compile-time. Note that such arrays can be considered to be homogeneous records.

9.3 Syntax
The type returned from a FUNCTION may be any type, rather than just a primitive.type.

   definition = { 1 , type } FUNCTION name ( { 0 , formal } )
               function.body
               |
             { 1 , type } FUNCTION name ( { 0 , formal } ) IS expression.list :
               (Note that this change is already subsumed in the changes permitting user defined types.)

   We also must change the syntax of an operand, so that functions can return arrays and records which can then be subscripted:
       operand = operand [ subscript ]
              (Note that Geoff proposed that expressions should be subscripted, rather than operands, but he has agreed that operands are better, because this reduces the need for `superfluous' brackets when used inside more complex expressions.)

9.4 Semantics
Same as FUNCTIONs returning scalar variables. All the aliasing rules still apply.

Since the sizes of the results are known at compile-time, the compiler can allocate temporaries if required.

Since a FUNCTION can return an object of array or record type, the function call may be subscripted directly.

9.5 Other points
It may be that we have to do some work to make this efficient: In many cases where a temporary might be used, without too much cost, for scalar types, this will be unsatisfactory for array types, because of the repeated copying which this implies. Therefore, the compiler must attempt to remove the temporaries.

An example might be a user routine written as follows:

    [n]INT FUNCTION double.array(VAL [n]INT arg)
    [n]INT temp :
    VALOF
      SEQ i = 0 FOR n
temp[i] := arg[i] * 2
RESULT temp
:
[n]INT x :
x := double.array[x]

In a naive implementation, this will involve assembling the result in temp, then copying this
to the actual result, which may also be a temporary, in the RESULT statement. There might then
be a final copy to the real result. In the example just given, this will require two copies.
Since the aliasing rules of our implementation of occam imply that the formal result parameter
of the call (which is passed as a pointer) cannot alias the formal parameter, the compiler is
permitted to remove temp entirely, and to use the formal result parameter directly. In the previous
example, this will remove one of the array copies.
A heuristic might be: if a local variable is declared inside the function, and is returned directly
as a RESULT, and the size of that variable is larger than some cutoff size, then the local variable is
removed and all references to it are replaced with references to the formal result variable.

10 Dropping FROM and FOR in segments

10.1 Description
Permit either the FROM phrase or the FOR phrase (not both) to be dropped from segments of arrays.

10.2 Rationale
Typographical convenience in the common cases where the start (FROM) value is 0, or the length
(FOR) value is the 'rest of the array'.

10.3 Syntax
\[
element \quad = \quad [ \text{element \ FROM \ subscript \ FOR \ count} ] \\
| [ \text{element \ FOR \ count} ] \\
| [ \text{element \ FROM \ subscript} ]
\]

\[
teble \quad = \quad [ \text{table \ FROM \ subscript \ FOR \ count} ] \\
| [ \text{table \ FOR \ count} ] \\
| [ \text{table \ FROM \ subscript} ]
\]

10.4 Semantics
When the FROM part is omitted, this is inferred to be constant zero.
When the FOR part is omitted, this is inferred to mean 'the rest of the array'. Thus
\[ [ \text{name \ FROM \ start} ] \] is equivalent to: \[ [ \text{name \ FROM \ start \ FOR ( SIZE name )} - \text{start} ] \]

11 RESHAPING arrays

11.1 Description
Provide a new operator RESHAPE, with similar syntax to RETYPES. Permit an array item to be
RESHAPEd into an array of the same base type, where one or more dimension has a size which is
variable, provided that the total size is still equal to the size of the right-hand-side. The number
of dimensions of the left-hand-side does not need to be identical to that of the right-hand-side.
11.2 Rationale
This allows 're-shaping' arrays into other shapes. This is useful when an array is available as a
'scratch area', but must be used as a temporary array of a different shape.

Note that the use of the word RESHAPE rather than RETYPES is because this construct does
not rely on bit pattern 'punning', etc, it relies purely on the (documented) layout of arrays as
contiguous regions of memory. As such, the use of RESHAPE does not make a program non-
portable.

11.3 Syntax

definition = specifier name RESHAPE element :

   = | VAL specifier name RESHAPE element :

   New keywords: RESHAPE.

11.4 Semantics
The compiler will calculate the size of the RESHAPE by multiplying together the sizes of all known
dimensions, whether they are constant or variable. One 'open' dimension is permitted.

   It is invalid if the base type of the new array is not the same as the base type of the old array.
   It is invalid if the size of the new array specified is not equal to that of the old.

As in the current implementation of RETYPES, it is an error if any dimension of an array cannot
fit in an INT. In some cases, this will require a run-time check.

12 Scoping of PROTOCOL tags

12.1 Description
The tags used in a variant PROTOCOL are currently included as globally visible names. Since these
cannot be used except in a communication, they could be scoped into a private namespace for
each PROTOCOL (aka RECORDs).

   This would be an upwards-compatible change, and would not affect any purely-occam code.

12.2 Rationale
This is proposed to simplify the writing of programs with many protocols, and to provide consist-
sity with record field names (see section 2).

   This proposal is included directly from Geoff Barrett’s work on occam 3. See section 6.4.4 of
Draft occam3 reference manual.

12.3 Syntax
No changes.

12.4 Semantics
PROTOCOL tags are not visible to the programmer, except in certain contexts:

   - As part of a channel output expression.
   - As part of a tagged channel input expression.
   - As part of a variant channel input expression.

   In each case, the scoping of the tag is determined from the protocol of that channel.
12.5 Other points
There is an existing undocumented ability to read PROTOCOL tag values as if they were BYTE constants. This will not be supported in occam 2.5.

13 Constant fold ‘single-line’ FUNCTIONs

13.1 Description
Specify that ‘single-line’ FUNCTIONs may be used where compile-time constants are required, when all parameters and free variables are compile-time constants.

13.2 Rationale
Permits better abstraction for the programmer.

13.3 Syntax
No changes.

13.4 Semantics
We must extend the definition of ‘compile-time constant’ to include appropriate (‘single-line’) FUNCTIONs.
(We probably have to first establish what the existing rules are for ‘compile-time constant’.)

14 INLINE routines

14.1 Description
The keyword INLINE may be used in conjunction with PROC and FUNCTION as a hint to the compiler that a routine should be inlined. The compiler is not required to actually support this, but it should accept the keyword.
This enhancement has previously been implemented in INMOS’s occam 2 compilers.

14.2 Rationale
Permits better abstraction for the programmer.

14.3 Syntax

```plaintext
definition  =  { 1 , type } [ INLINE ] FUNCTION name ( { 0 , formal } )
                      function.body
                      :
                       | { 1 , type } [ INLINE ] FUNCTION name ( { 0 , formal } ) IS expression.list :
                       | [ INLINE ] PROC name ( { 0 , formal } )
                       procedure.body
                      :
New keywords: INLINE.
```
14.4 Semantics

The compiler is advised by the programmer that this routine is a good candidate for ‘inline expansion’ wherever that routine is called. The semantics of the call are otherwise identical to any normal call.

Note that the declaration of the routine is marked with the keyword, but each call site is affected.

15 Channel RETYPEing

15.1 Description

Channels may be RETYPEd to and from channels of a different protocol.

This enhancement has previously been implemented in INMOS’s occam 2 compilers.

15.2 Rationale

This is sometimes required when accessing specific hardware, in a similar manner to data RETYPES.

15.3 Syntax

No changes.

15.4 Semantics

A channel may be RETYPEd to another channel of a different protocol. VAL RETYPES are not permitted.

16 Channel constructors

16.1 Description

Arrays of channels may be constructed out of a list of other channels.

This enhancement has previously been implemented in INMOS’s occam 2 compilers.

16.2 Rationale

This makes it easier to use ‘generic’ routines which manipulate arrays of channels, from existing individual channels.

16.3 Syntax

No changes.

16.4 Semantics

A channel constructor creates an array of channels out of existing channels. The array may be subscripted, etc, and used like any other array of channels. All channels in the array must be used in the same direction.
17 PROTOCOL name IS ANY

17.1 Description
The anarchic protocol ANY may now be specified as part of a named, sequential, or variant PROTOCOL. It indicates that a single data item of any type may be communicated as part of that protocol.
This enhancement has previously been implemented in INMOS’s occam 2 compilers.

17.2 Rationale
This permits a programmer to distinguish effectively and securely between two different anarchic protocols, by giving them different protocol names.

17.3 Syntax

    simple.protocol = ANY

17.4 Semantics
A single communication of type ANY on a channel as part of a larger communication of named protocol is treated exactly like that of a CHAN OF ANY, as described in section 4.3.5 of the occam 2 Reference Manual.

18 Allocations

18.1 Description
Allocations must be written immediately after the declaration to which they refer.
Note: This is a change from strict occam 2 behaviour. However, a minor source code edit will always permit the code to be accepted.

18.2 Rationale
The occam 2 Reference Manual does not place any constraints on the location of an allocation when it described them in section A.3. We propose to enforce that the allocation is written immediately after its referenced declaration. This ensures that a compiler can process the allocation before processing any uses of that declaration.

18.3 Syntax
No change.

18.4 Semantics
An allocation must occur immediately after the declaration to which it refers. The only intervening syntactic entities which may occur are other allocations of names declared in the same declaration.

19 Counted array input

19.1 Description
The semantics of counted array input are not exactly the same as those described in the occam 2 Reference Manual. A counted array input of the form
message ? len :: buffer

is no longer permitted to refer to the len as part of the expression buffer.
Note: This is a change from strict occam 2 behaviour.

19.2 Rationale

This makes it possible to treat the input as a single input, which communicates the data and the length of the data as a single item.

19.3 Syntax

No change.

19.4 Semantics

message ? len :: buffer

This input receives an integer value which is assigned to the variable len, and that number of components, which are assigned to the first components of the array buffer. The assignments to len and buffer happen in parallel and therefore the same rules apply as for parallel assignment. That is, the name len may not appear free in buffer, and vice versa.

19.5 Backwards compatibility

As a concession to backwards compatibility, the compiler permits communications of the form:

channel.exp ? name :: [ array.exp FROM 0 FOR name ]

These are transformed into the equivalent form:

cchannel.exp ? name :: array.exp

20 Future directions

20.1 CHAN OF ANY

CHAN OF ANY should be considered obsolete; it should be replaced by a named PROTOCOL (which may be of type ANY), to give greater security. See section 17.

20.2 Counted array input

The ability to name the length of a counted array input as the length of the receiving buffer is obsolescent. The programmer should rewrite the statement using the equivalent form. See section 19.5.