An occam2.1 Checklist

Computing Laboratory, University of Kent at Canterbury, CT2 7NF.

1. Introduction

1.1. Top-Down View

An *occam* program is a `<process>`.

A `<process>` is a (possibly null) sequence of `<declarations>` followed by an `<executable>`. Each `<declaration>` starts on a separate line and at the same level of indentation. The `<executable>` also starts at this same level of indentation. The scope of any `<declaration>` covers only those `<declarations>` and the `<executable>` that follow it.

An `<executable>` is either a `<primitive.process>` or a `<structured.process>`.

1.2. Bottom-Up View

Occam processes `INT`s (16, 32 or 64 bits, depending on the word-length of the target processor), `INT16`s (16 bits, regardless of the word-length of the target processor), `INT32`s (32 bits, also regardless ...), `INT64`s (64 bits, also regardless ...), `BYTE`s, (8 bit unsigned numbers — i.e. 0..255), `BOOL`s and ANSI/IEEE-754 standard floating-point numbers in two precisions — `REAL32` and `REAL64`. Variables, channels and expressions are all typed and strong checking rules are enforced that do not allow inconsistent mixes. Character literals (in single quotes) are treated as `BYTE`s and character strings (in double quotes) as fixed length `BYTE` arrays. The following operators are defined for expressions:

`+, -, *, /, \`	(operands `INTxx`, result `INTxx`)
`PLUS, MINUS, TIMES`	(operands `INTxx`, result `INTxx`)
`AFTER`	(operands `INTxx`, result `BOOL`)
`+, -, *, /`	(operands `REALxx`, result `REALxx`)
`<, <=, >, >=`	(operands `INTxx`, `REALxx` or `BYTE`, result `BOOL`)
`AND, OR, NOT`	(operands `BOOL`, result `BOOL`)
`=, <>`	(operands same type, result `BOOL`)
`/\ \, ~, ><`	(operands `INT`, result `INT`)
`<<, >>`	(left operand `INT`, right operand small positive `INT`, result `INT`)

Warning: there is no precedence ordering for operators nor any 'left-to-right' evaluation rule — use brackets!

Note: infix operators with `INTxx` or `REALxx` operands must have the same precision (i.e. 16, 32 or 64 bits), as well as the same type, on each side. They return values of that same precision and type.

Note: to convert between types/precisions, just use the target type/precision name as a prefix operator on the source type/precision value — see section 11.5.

Note: `PLUS`, `MINUS`, `TIMES` and `AFTER` are best applied to time values — see sections 2.4, 3.4, 3.5 and 4.3. Time cycles through the complete range of `INT`s. If we `PLUS` one to the most positive `INT`, we get the most negative `INT` — not an overflow error.

2. Declarations

2.1. Constant Abbreviation

```
VAL INT max IS 50:
VAL INT double.max IS 2*max:
VAL BYTE first.letter IS 'A':
VAL []BYTE hello IS "Hello World*c*n":
VAL BYTE control IS #1F:
VAL BYTE bell.char IS control /\ 'G':
VAL [8]INT masks IS [#01, #02, #04, #08, #10, #20, #40, #80]:
```

Note: these are a special form of the more general *occam* concept of abbreviation — see section 8.

2.2. Variable Declaration

```
INT a, b:
[max]INT c:
[double.max][max]BYTE d:
```

The array `c` has `max` elements indexed `c[0]` to `c[max-1]` inclusive. The declared size of an array must be a constant. The variables `a` and `b` and elements of `c` are `INT` quantities. The array `d` has `double.max` elements indexed `d[0]` to `d[99]`. The elements of `d` are `[max]BYTE` arrays. The byte `d[a][b]` is the `b`th. element of the `a`th. element of `d`.

2.3. Channel Declaration

```
CHAN OF BYTE p:
[max << 2]CHAN OF INT q:
```

Here, `p` is a channel carrying `BYTE` values and `q` is an array of 200 `INT` channels indexed `q[0]` to `q[199]` inclusive.

2.4. Timer Declaration

```
TIMER tim:
```

To access the clock, `INT` values need to be input from a `TIMER` (see Section 3.4). Any number of `TIMER`s may be declared — they all go at the same speed (e.g. one tick per microsecond for high-priority processes on a transputer), but they are not obliged to show the same time.

2.5. Process Abstraction

```
PROC foo (VAL []BYTE s, VAL BOOL mode, INT result,
          CHAN OF INT in, out, CHAN OF BYTE pause)
  <process>
:
```

VAL <*type*> <*formal*> parameters are similar to value parameters in *Pascal* — except that they are treated as constants within the <*process*> body and may not be altered. <*type*> <*formal*> parameters are similar to reference (i.e. `var`) parameters in *Pascal*.

Note: *occam* parameter mechanisms are formally defined in terms of the *occam* abbreviation concept — section 8. Aliasing of item names cannot be introduced through parameters (as can happen in *Pascal*, *C*, *FORTRAN*, *Ada*, *Modula-2*, ...).

The `<process>` body of the `PROC` is as defined in section 1.1 above — i.e. it can have private declarations. This body must be indented exactly two spaces from the `PROC` line. Global names may be accessed directly from the body — but please, of course, avoid global variables and channels! Parameter channels must be used exclusively for input or exclusively for output within the body. In `foo`, the parameter `s` may be instanced either with a `BYTE` array variable (like `d`) or a `BYTE` array constant (like `hello`). `PROC`s may be used to create autonomous processes *or* for traditional procedure calls.

Warning: `PROC`s may not have recursive definitions — sorry!

3. Primitive Processes

3.1. Assignment Process

```
a := c[2] + b
```

Occam expressions have no side-effects! Because of the anti-aliasing laws (see section 8), different names in the same scoping context always refer to different items — no nasty surprises! This gives us *referential transparency*, a posh name for *what-you-see-is-what-you-get*. Such good behaviour is not displayed by *any* of the traditional languages for procedural programming (*Pascal*, *C*, *FORTRAN*, *Ada*, *Modula-2*, ...) and is a major cause of hard-to-find and serious error.

3.2. Input Process

```
in ? a
```

This process is suspended until a message arrives on channel `in`. It then inputs the contents into the variable `a` (whose type, of course, must match that of the channel).

3.3. Output Process

```
out ! a + (2 * b)
```

The expression is evaluated and output down channel `out` (whose type, of course, must match that of the expression). This process is then suspended until the message is accepted by the receiving process (at the other end of the channel).

Every communication between parallel processes uses this synchronisation mechanism — sometimes called a ‘rendezvous’.

3.4. Timer Process

```
tim ? t
```

The result of this process is that the value of the timer `tim` is assigned to the variable `t` (which must be an `INT`).

Although the syntax for reading them is similar, timers are different from channels. Channel input is *synchronised* — we have to wait for the message to arrive! Data must not get lost during channel communication!!

On the other hand, timers behave like *un-synchronised* 'channels' — when we read the time, we just want the latest value. We don't want to have to wait for the clock to tick! Data gets lost — all those time-values we didn't read — but that's OK for this kind of device. We don't want the clock to stop just because we don't look at it!!

3.5. Timeout Process

```
tim ? AFTER t
```

This process is suspended until the value on the `tim` timer (always available and increasing) gets `AFTER` the value of `t`. Note that `t` may be any `INT` expression, but don't forget to use `PLUS` and `MINUS` when working with time values (rather than + and −).

3.6. Null Process

```
SKIP
```

This process immediately terminates. It's needed because sometimes the syntax requires a process and we have nothing to do. Of course, this process generates no object code!

3.7. Suspend Process

```
STOP
```

This process is suspended and never resumed — useful for panic situations.

4. Structured Processes

4.1. Sequential Processes

```
SEQ
  <process.0>
  <process.1>
  <process.2>
```

The `<processes>` are as described in 1.1 (i.e. may have private declarations) and must be indented exactly two spaces further than the `SEQ` keyword. The extent of the `SEQ` construct is finished when the indentation level returns to the same as (or less than) that of the `SEQ`.

The construct means that `<process.0>`, `<process.1>` and `<process.2>` are to be executed in that order. Any channel used within a `SEQ` construct must be used exclusively for input or exclusively for output.

4.2. Parallel Processes

```
PAR
  <process.0>
  <process.1>
  <process.2>
```

The indentation and extent rules are as for the `SEQ` above. The construct means that `<process.0>`, `<process.1>` and `<process.2>` are to be executed in parallel. This construct does not terminate until all its component processes have terminated. The order in which the `<processes>` are listed is irrelevant.

Parallel processes may not assign or input to shared data (but they may inspect shared data). Parallel processes may not share input channels. Parallel processes may not share output channels. The effect of these rules is that parallel processes can only influence each other by communicating along dedicated point-to-point channels. So long as there are no ALTs (see section 4.3) present, their semantics — i.e. the services they provide to, and demand from, their environment — are completely deterministic, regardless of internal scheduling patterns or relative processor speed (when running on multiple processors).

4.3. Non-Deterministic Process

```
ALT
  <guard.0>
    <process.0>
  <guard.1>
    <process.1>
  <guard.2>
    <process.2>
```

Each <*guard*> is indented two spaces and its associated <*process*> is indented a further two spaces. We shall refer to such a pair as a <*guarded.process*>. The ALT construct is suspended until one or more of the <*guards*> becomes ‘ready’ (see below). One of the ‘ready’ <*guards*> (chosen *arbitrarily*) is then executed followed by its associated <*process*>. The order in which these <*guarded.processes*> are listed is irrelevant.

A <*guard*> is either a <*simple.guard*> or a <*pre.conditioned.guard*>.

A <*simple.guard*> is most commonly an <*input.guard*> :–

```
in ? x
```

This becomes ‘ready’ if a message is pending on the channel. Its execution is to input the message.

Alternatively, a <*simple.guard*> may be a <*time.out.guard*> :–

```
tim ? AFTER absolute.time.out
```

This becomes ‘ready’ when the value from the tim timer (always available and increasing) becomes AFTER the absolute.time.out value. Its execution is null.

Thirdly, a <*simple.guard*> may be a <*null.guard*> :–

```
SKIP
```

This is always ‘ready’ and has a null execution.

Finally, a <*pre.conditioned.guard*> has the form :–

```
<pre.condition> & <simple.guard>
```

where the <*pre.condition*> is any BOOL expression. If the <*pre.condition*> is FALSE, the <*simple.guard*> and its associated <*process*> are not candidates for execution — even if the <*simple.guard*> becomes ‘ready’. (Note that, because of the rules forbidding shared data between parallel processes, the value of any <*pre.condition*> cannot alter whilst awaiting a <*simple.guard*>.)

Note: in fact, a SKIP guard is compelled to have a pre-condition. To get an ‘un-conditioned’ SKIP guard, use TRUE & SKIP. But beware: TRUE & SKIP only makes sense within a PRI ALT — see section 6.

4.4. Sequential Choice

```
IF
  <condition.0>
    <process.0>
  <condition.1>
    <process.1>
  <condition.2>
    <process.2>
```

The indentation and extent rules are as for the `ALT` construct above. We shall refer to a *<condition>* and its associated *<process>* as a *<conditional.process>*. This corresponds to a traditional :–

```
if <condition.0> then <process.0> else
if <condition.1> then <process.1> else
if <condition.2> then <process.2> else STOP
```

of, say, *Pascal*. Note the `STOP` if no *<condition>* holds!

4.5. Sequential Loop

```
WHILE <condition>
  <process>
```

This is just like *Pascal*. The *<process>* must be indented two spaces from the `WHILE`.

There are no `GOTO`s in *occam* nor any abnormal loop breaking mechanisms. Thus, we always know that, when the loop does terminate, the*<condition>* must be `FALSE`. This is not the case for *Pascal*, *C*, *FORTRAN*, *Ada*, *Modula-2*, ...

4.6. Process Instantiation

```
foo (hello, FALSE, a, q[0], q[199], p)
```

This is like a procedure call in *Pascal*. Value parameters must be instanced with expressions of the correct type. Reference parameters must be instanced with variables of the correct type. Channel parameters must be instanced with channels of the correct protocol (section 7). We may also have timer parameters and port (section 10) parameters.

5. Replicated Processes

The `SEQ`, `PAR`, `ALT`, and `IF` constructs may be replicated. Suppose *XXX* is one of these four keywords. Then :–

```
XXX i = start FOR n
  <thing.which.may.use.i>
```

is short-hand for :–

```
XXX
  <thing.with.i.replaced.by.start>
  <thing.with.i.replaced.by.(start + 1)>
  ...
  <thing.with.i.replaced.by.(start + (n - 1))>
```

For `SEQ` and `PAR`, the <*thing.which.may.use.i*> is a <*process*>. For `ALT`, it is a <*guarded.process*>. For `IF`, it is a <*conditional.process*>. The replicator control variable, `i`, is declared by this construct. It has scope for, and may not be altered by, the <*thing.which.may.use.i*>. For a replicated `PAR`, the replication factor, `n`, must be a constant.

The replicated `SEQ` corresponds to a traditional `for`-loop. The replicated `IF` is an efficient construct for performing a bounded length search. The <*conditional.process*> of an `IF` construct can itself be an `IF` construct. The <*conditional.processes*> of an inner `IF` are treated as through they were at the same level as those of the outer `IF`. Thus, a search through the array `c` for a particular item, say `42`, may be coded :–

```
IF
  IF i = 0 FOR SIZE c
    c[i] = 42
      found.it, its.index := TRUE, i
  TRUE
    found.it := FALSE
```

where `SIZE c` is the number of elements in `c`. This could be coded using a `WHILE` or replicated `SEQ` construct — but it would be much less clear and efficient.

Similarly to `IF` constructs, the <*guarded.process*> within an `ALT` construct can itself be an `ALT` construct — the <*guarded.processes*> of the inner `ALT` being treated as though they were at the same level as those of the outer `ALT`. The use of replicated `ALT` and `PAR` constructs has no analogy within traditional programming languages.

The expression `[c FROM start FOR n]` represents a 'slice' of the array `c` from element `c[start]` to `c[start + (n - 1)]` inclusive. Array slices may be assigned to one another, or input or output through channels. The sizes of slices on either side of the assignment or channel must be equal. Such operations are more efficient than a single assignment, input or output controlled by a corresponding replicated `SEQ`. Array parameters may also be instanced with array slices.

6. Prioritised Processes

We may write `PRI ALT` instead of `ALT`. The only difference is that if more than one of the <*guards*> is 'ready', the choice of which one to execute is not *arbitrary* but is the *first* in the order listed.

We may write `PRI PAR` instead of `PAR`. This imposes a 'priority' on the <*processes*> to be run in parallel according to the order in which the <*processes*> are listed. Lower priority <*processes*> are never executed if there is a higher priority one able to run. The current transputer implementation restricts the number of components of a `PRI PAR` to just two — 'high' priority and 'low' priority.

7. Protocols

All channel declarations state the message structure (called the `PROTOCOL`) carried by that channel. All messages sent over a channel must conform to that protocol. All protocols used so far have been 'simple' ones — i.e. *occam* data-types. However, we frequently want messages with richer structures — e.g. a mixture of data-types, differing data-types (depending upon the run-time state) or differing amounts of the same data-type (depending upon the run-time state).

7.1. Simple Protocols

Any *occam* data-type may be used for a channel protocol — even arrays:

```
CHAN OF [max]REAL64 chunk:
```

Suppose `ping` is a `[max]REAL64` array. Then:

```
chunk ! ping
```

outputs all 50 elements of `ping` down the channel `chunk`. At the other end:

```
chunk ? pong
```

inputs all 50 elements into `pong` (which must, of course, be a `[max]REAL64` variable).

Note: this single transfer of the whole array is much more efficient (in terms of code space and execution speed) than the replicated transfer of its elements, one at a time, down a `REAL64` channel.

7.2. Sequential Protocols

Suppose we need to send three values of differing types — say an integer, a real and a boolean — from one process to another. First, define a suitable message structure:

```
PROTOCOL PACKET IS INT; REAL64; BOOL:
```

where `PACKET` is our own choice of name. Then, we may declare:

```
CHAN OF PACKET carry:
```

When the channel `carry` is used, we must output a message that matches the structure of the `PACKET` — for example:

```
carry ! n; a*(b + c); flag
```

where `n` must be an `INT`, `a`, `b` and `c` must be `REAL64`s and `flag` must be a `BOOL`. As always, this output process is suspended until the whole message has been input at the other end of the channel.

The receiver must provide variables that also match the `PACKET` protocol — for example:

```
carry ? i; x; mode
```

where `i` must be an `INT`, `x` must be a `REAL64` and `mode` must be a `BOOL`.

There is an analogy in the way that a sequential protocol packages messages to the way that a 'record' structure packages data-types. However, it is a little bit different since the ordering of message fields is significant — the data is delivered in the sequence defined by the protocol declaration. We may safely rely on this ordering, for example, to send information in the first part of the message that defines where later parts are to be stored:

```
carry ? i; x[i]; mode[i]
```

where `x` and `mode` must now be (respectively) `REAL64` and `BOOL` arrays.

7.3. Variant Protocols

If we want to send different message structures at different times down the same channel, we use a variant (or `CASE`) protocol. In the protocol declaration, we list the various message structures we wish to send, each preceded by a unique 'tag' name of our choice. For example:

```
PROTOCOL SERVICE
  CASE
    enquire; INT
    update; INT; REAL32
    reset
    terminate
:
```

The tags `enquire`, `update`, `reset` and `terminate` are new and distinct constants of a new and private type. They are introduced by the above declaration. They can only be used in communications over channels carrying this protocol — for example:

```
CHAN OF SERVICE to.server:
CHAN OF REAL32 from.server:
```

Notice that two of the tags (`reset` and `terminate`) are followed by no further message. The tag is the only information conveyed.

Outputting down variant channels is straightforward — for example:

```
SEQ
  to.server ! reset
  ...
  to.server ! update; 42; 3.142
  ...
  to.server ! enquire; 42
  from.server ? x
  ...
  to.server ! terminate
```

where `x` is a `REAL32` variable. The inputting side uses a `CASE` syntax to distinguish between the variant message structures according to the tag — for example:

```
WHILE running
  to.server ? CASE
    enquire; i
      from.server ! B[i]
    update; i; B[i]
      SKIP
    reset
      ...  clear B back to initial values
    terminate
      running := FALSE
```

where `running` is a `BOOL`, `i` is an `INT` and `B` is an array of `REAL32`s.

Note: if the system is in a state where some of the protocol variants will not be sent, the inputting process is not obliged to cater for them (i.e. we may omit the relevant tagged input lines and associated processes). If that leaves only one variant to be serviced, a short-hand form of the input syntax may be used — for example:

```
  to.server ? CASE update; i; B[i]
```

Note: if a variant arrives that is not allowed for by the inputting process, this is a run-time error and the process `STOP`s. This is caused, of course, by a system design error.

7.4. Counted Array Protocols

A common form of message is a list of data items whose length is determined at run-time. Such a structure is called a *counted array* and is described by the following syntax:

```
<count.type>::[]<any.type>
```

where *<count.type>* is either a BYTE or an INT and *<any.type>* is any *occam* data type. It defines a message with two components — a count and an array (whose size is that count). The count may be zero, but must not be negative.

A counted array may be a protocol on its own or a field in a sequential or variant protocol. For example:

```
PROTOCOL STRING IS BYTE::[]BYTE:
PROTOCOL FLEXI.CHUNK IS INT::[]REAL64:
```

Then, we may declare:

```
CHAN OF STRING screen:
CHAN OF FLEXI.CHUNK flexible:
```

STRING channels allow strings of up to 255 characters (including the null string) to be transmitted. For example:

```
VAL []BYTE epitaph IS "Goodbye Cruel World ...":
VAL BYTE size IS BYTE (SIZE epitaph):
screen ! size::epitaph
```

This outputs the entire epitaph. As always, this output process does not complete until the message is input by the process at the other end of the channel.

We do not have to output the whole of an array — for example:

```
flexible ! n::ping
```

outputs only the first n elements of the REAL64 array ping, where n is an INT value. If ping has less than n elements, this is a run-time error.

At the receiving end:

```
screen ? length::text
```

inputs the BYTE count of the STRING message into length (which must be a BYTE variable) and the array part into the first length elements of text (which must be a sufficiently long BYTE array).

When we need to communicate to and from sections of an array *other than* the initial one, we must use array slices (see the end of section 5). Consider:

```
flexible ? m::[pong FROM start FOR slice]
```

where m is an INT variable, start and slice are INT values and pong is a REAL64 array with at least (start + slice) elements. This process inputs the FLEXI.CHUNK count value into m. The array part of the message goes into the m consecutive elements of pong starting from index start. There will be a run-time error unless $0 \leq m \leq$ slice.

8. Abbreviations

8.1. Reference Abbreviations

Any *occam* item (i.e. a piece of data, a channel, a timer or a port) may be passed as an *actual* parameter to a (matching) *formal* parameter of a PROC. During the execution of the PROC body, the item acquires the formal parameter name as an 'alias' or 'abbreviation'.

Aliasing (i.e. allowing one item to be referred to by different names) is uncontrolled in traditional programming languages and leads to semantic complexities that are generally underestimated, easily overlooked and cause serious error. In *occam*, when a new name is introduced for an existing item, we are only allowed to use the new name — the old name is banned! We have, therefore, the assurance that different names in the same scope refer to different items — regardless of the context in which they appear. This may seem a minor issue, but it is highly significant in practice.

We shall return to parameter mechanisms shortly. First, *occam* has a direct way of introducing a new name for an existing item:

```
<item.specifier> <new.name> IS <old.name>:
```

where <*item.specifier*> indicates a data-type, channel-type, timer or port-type. The <*old.name*> may be a variable, array element or array slice. It names a particular item conforming to the <*item.specifier*>. The <*old.name*> may not be used whilst the <*new.name*> is in scope. Any variables used in <*old.name*> to determine individual array elements or slices may be used in the scope of <*new.name*>, but may not be changed by it.

The <*new.name*> is said to be an 'abbreviation' of the <*old.name*>. Such an abbreviation may occur anywhere a declaration is allowed. As with declarations, its scope covers only the process that immediately follows it. Of course, that process may have its own declarations and abbreviations. For example:

```
INT result IS n:
[]REAL64 row IS X[i]:
CHAN OF FLEXI.CHUNK out IS flexible:
<process>
```

where n must be an INT variable, X must be a REAL64 matrix, i must be an INT value (within the range of the first dimension of X) and flexible must be a CHAN OF FLEXI.CHUNK.

Within <*process*>, the 'old' names n, X[i] and flexible may not be used — we must use the 'new' names result, row and out instead. The value of i may be used but not changed! More generally, if i were an expression, all its constituent variables would have their values frozen. Disjoint parts of the matrix X may be referenced (e.g. X[j]), but only if we can guarantee their disjointness — e.g. i and j are constant values, known to be different at compile time.

We may, of course, abbreviate abbreviations:

```
[16]REAL64 section IS [row FROM start FOR 16]:
```

where the value of start is now frozen within the scope of section. Note that section[0] corresponds to row[start] which corresponds to X[i][start], but that the latter two names may not now be used.

8.2. Value Abbreviations

Any data *value* (i.e. an expression made from variables, literal constants, array elements and array slices) may be passed to a (matching) formal VAL parameter of a PROC. During the execution of the PROC body, the value acquires the formal parameter name. We may not assign or input to this name — it represents a *value*, not a *variable*, and we are not allowed to change it!

Again, *occam* has a direct way of introducing names for data values:

```
VAL <data.type> <new.name> IS <expression>:
```

We may not assign or input to <*new.name*>. Any variables mentioned within the <*expression*> can continue to be used within the scope of <*new.name*>, but may not be changed by that process. The <*expression*> must yield a value of <*data.type*>.

Section 2.1 uses this mechanism to name values known at compile-time (i.e. constants). However, values computed at run-time may also be given names — for example:

```
VAL REAL32 hypoteneuse IS SQRT ((a*a) + (b*b)):
VAL []REAL64 row IS X[i]:
VAL n IS SIZE row:
<process>
```

where a and b must be REAL32 values, X must be a REAL64 matrix and i must be an INT value. Within <*process*>, the 'new' names hypoteneuse, row and n represent values that cannot be altered. The names a, b, i and X[i] may be freely used, but again their values cannot be altered.

Note: value abbreviations *do* allow multiple names for the same piece of data to be introduced (e.g. when the <*expression*> is just a variable) and used in the same scope — but only after first 'freezing' that piece of data to a constant value. Different names for the same constant do not lead to the semantic complexities of different names for the same variable.

8.3. Parameters and Abbreviations

Occam PROC calls are formally defined by in-line expansion into the text of the PROC body. Formal parameters are linked to the actual ones by a sequence of abbreviations. For example, the call:

```
foo (hello, FALSE, a, q[0], q[199], p)
```

(see section 2.5 and 4.6) transforms into:

```
VAL []BYTE s IS hello:
VAL BOOL mode IS FALSE:
INT result IS a:
CHAN OF INT in IS q[0]:
CHAN OF INT out IS q[199]:
CHAN OF BYTE pause IS p:
<process>
```

where <*process*> is the text of the body of foo.

The point of this relationship is that it enables us to derive anti-aliasing laws regarding the use of parameters directly from those for abbreviations. These laws are strictly enforced.

For example, if the actual parameter `a` was in scope at the point of definition of `foo` and if it was mentioned in its body, the above call would lead to a violation of the aliasing rules for reference abbreviation and would be rejected by the compiler.

Similarly, the calls:

```
foo (hello, FALSE, a, q[0], q[199], panic[a])
```

where `panic` is an array of `BYTE` channels, and:

```
foo (hello, FALSE, a, q[42], q[42], p)
```

would always be suppressed.

Note: free names in `PROC` bodies are bound to the global definitions visible at the point of the `PROC` definition — not at the `PROC` call. The above transformation is, therefore, subject to the condition that any such free names do not have re-declarations in scope at the point of the call. If those names are so re-declared, the re-declarations must be eliminated (by choosing different names that cause no similar clashes) before the transformation becomes valid. This follows precisely the rule for consistent substitution from the λ-calculus. [This rule is relevant only for completing the formal definition of the `PROC` call through in-line substitution of the `PROC` body (i.e. β-reduction). In practice, of course, *occam* is not usually implemented like this — i.e. we do not have to avoid such re-declarations.]

8.4. And So ...

The anti-aliasing laws greatly simplify the semantics and implementation of parameter passing. For reference parameters, *copy-in/copy-out* and *call-by-address* mechanisms are indistinguishable — the compiler may apply whichever is most efficient (e.g. the former for 'small' items and the latter for 'large' ones). For value parameters, *copy-in* and *call-by-address* are also the same — so it is perfectly secure to pass arrays by address (since the `PROC` body is forbidden to update them).

To illustrate the simplicity that results from *occam*'s careful control of aliasing, consider:

```
SEQ
  n := n + a
  n := n - a
```

where `n` and `a` are `INT` variables. Assuming we are not dealing with values that cause arithmetic overflow, everything we know about algebra and the properties of variables, assignment and sequencing tells us that the above code changes nothing. With *occam*, this simple interpretation is the correct one and the code may be safely replaced with a `SKIP`.

With languages that take a less rigorous approach towards aliasing (such as *Pascal*, *C*, *FORTRAN*, *Ada*, *Modula-2*, ...), the semantics are not so straightforward. We have to look into the context of the code and check the bindings of the variables. If both `n` and `a` refer to the same data item (e.g. when `n` and `a` are formal reference parameters to which the same actual parameter has been supplied), the above code changes that data item to zero! The problem is that the `a` 'variable' is no longer behaving in the way we expect a variable to behave — i.e. remain the same unless we change it. If `n` is an alias name for `a`, the value of `a` is different in each of the above lines!!

Such semantic complexities are easily missed when reasoning about algorithms in traditional languages. With *occam*, different names in the same scope refer to different items whatever the context and no such singularities of meaning are possible. What you see is what you get — end of story!

9. Retyping

It is sometimes helpful to view a piece of data as though it had a different type structure from its ‘natural’ one. *Occam* allows this with a slight variation on its abbreviation concept. For example, we may have reference retyping:

```
<data.type> <new.name> RETYPES <old.name>:
```

All the anti-aliasing rules for reference abbreviation apply. The only difference is that the type of <*old.name*> need not be the same as <*data.type*>, but its representation must occupy the same number of bits. Using <*new.name*>, we manipulate the bit-string value of <*old.name*> as though it belonged to the new <*data.type*>. No type-conversion takes place — the interpretation of the value in the new type is implementation-dependent upon the representation formats for the old and new types.

For example, if `n` is an `INT32` variable, then:

```
[4]BYTE b RETYPES n:
b[0], b[1], b[2], b[3] := b[3], b[2], b[1], b[0]
```

reverses the ordering of the byte representation of `n` (which may be useful in reformatting data produced by a ‘little-endian’ machine for use in a ‘big-endian’ one).

Consider also:

```
REAL64 x RETYPES [buffer FROM speed.index FOR 8]:
x := speed
```

where `buffer` is a `BYTE` array with at least `(speed.index + 8)` elements and `speed` is a `REAL64`. This packs the `speed` value into an 8-byte slice of the array `buffer` (prior, perhaps, to output down an unstructured byte-stream).

We also have value retyping:

```
VAL <data.type> <new.name> RETYPES <expression>:
```

Again, the same rules for value abbreviation apply to value retyping. Again, the type of <*expression*> need not be the same as <*data.type*>, but it must have the same sized representation.

For example:

```
VAL REAL64 x RETYPES [buffer FROM speed.index FOR 8]:
speed := x
```

retrieves the `REAL64` value (packed as an `[8]BYTE` slice of `buffer`) into a `REAL64` variable. Notice that the original packing could also have been done with a value retyping:

```
VAL [8]BYTE s RETYPES speed:
[buffer FROM speed.index FOR 8] := s
```

and, indeed, the unpacking could have been done by reference:

```
[8]BYTE s RETYPES speed:
s := [buffer FROM speed.index FOR 8]
```

Retypings are implementation-dependent in their meaning and should be used with care and restraint! Generally, their scope should be very localised — they only extend over one line in the above examples.

Occasionally, value retypings are used to define (global) constants — i.e they have a very long scope. For example:

```
VAL REAL64 pi RETYPES #400921FB54442D18 (INT64):
VAL REAL64 infinity RETYPES #7FF0000000000000 (INT64):
VAL REAL64 NaN RETYPES #7FF0000200000000 (INT64):
```

(where the # symbol introduces a hexadecimal literal) defines π, ∞ and *Not-a-Number* with full 64-bit precision according to the IEEE/ANSI-754 standard format. Expressing π as a `REAL64` literal:

```
VAL REAL64 pi IS 3.1415926535897932384626433832 80:
```

relies on conversion routines in the compiler that are accurate to the last bit. The other two values, `infinity` and `NaN`, cannot be so expressed at all!

10. Ports

External devices that provide/receive information via memory-mapped registers are interfaced to *occam* processes through the notion of *ports*.

Such a register must not be modelled by a variable, since it does not behave as a variable — consecutive reads will often yield different values. It must not be modelled by a channel, since read/write access is not synchronised with a matching write/read.

In fact, ports are asynchronous channels. They have a channel-like syntax for declaration and use:

```
PORT OF INT16 control.register:
PLACE control.register AT #80000100:

PORT OF INT16 data.register:
PLACE data.register AT #80000104:
```

Ports may be declared with any *occam* data-type. The `PLACE`ment line defines the base address (usually expressed as a hexadecimal `INT` literal) of the data-structure representing the port.

Warning: although these base addresses must be complie-time values, current *occam* compilers make no attempt to check that user-placed structures do not conflict either with each other or with the compiled code, process workspace or any ‘special’ locations peculiar to the target architecture! We are on our own here!!

The following code busily polls the `control.register` until its third bit is set, whereupon it outputs a `value` (which must be an `INT16`) to the `data.register`:

```
VAL INT bit.3 IS 4:  -- bit mask
INT16 control:
SEQ
  control.register ? control
  WHILE (control /\ bit.3) = 0
    control.register ? control
  data.register ! value
```

Note: ‘busy’-polling is not generally a good idea — the loop should at least have a decent ‘sleep’ process in its body. Much better than any such *polling* of a control register is to *wait* for an ‘interrupt’ signal telling us to proceed — see section 11.6

Input and output on ports are asynchronous operations — i.e. they never get blocked and they always terminate. They are usually implemented as normal *loads* and *stores* on memory. Reading from a `TIMER` is logically similar to reading from a `PORT OF INT`. Note that, as happens with timers (but unlike channels), ports introduce non-determinism into the semantics.

Ports should only be used to interface *occam* systems to non-*occam* devices. This means that a port is used by an *occam* system only at one end! Ports may not be used to provide asynchronous communication between parallel processes *within* the system!!

11. Odds and Ends

This section summarises the facilities in the language not so far mentioned. They provide important abstractions that help ease the expression of algorithms, but provide no 'new' concepts that would be unfamiliar to users of traditional languages (although, of course, *occam* enforces a very clean and secure binding for them). They were not introduced earlier because their details would have caused an unnecessary distraction.

11.1. Parallel Assignment

We have seen examples of parallel assignment in code fragments in sections 6 and 9. The full syntax is:

```
<list.of.variables> := <list.of.expressions>
```

where a <*variable*> is anything assignable (i.e. a variable, array element or array slice) and an <*expression*> is anything that represents a data value. The lists are comma-separated, have equal length and the types of corresponding elements must match.

The semantics of parallel assignment has two consecutive phases:

```
SEQ
  ...  evaluate the <list.of.expressions>
  ...  assign to the <list.of.variables>
```

Each phase operates in parallel. Since *occam* expressions cause no side-effects, no restrictions are imposed on the <*list.of.expressions*>. On the other hand, the <*list.of.variables*> must be distinct and independent. Thus:

```
i, i :=  3, 4
```

is not allowed and neither is:

```
i, a[i] :=  3, 4
```

On the other hand, swapping the values of two variables:

```
i, j :=  j, i
```

is always (thanks to the anti-aliasing laws) legal — as is any permutation of a list of variables (e.g. see section 9).

11.2. Functions and Value Processes

A <*value.process*> is an *occam* process that yields a (list of) data values. Because it is only used as (part of) an expression, it must cause no side-effects and be fully deterministic. Thus, it may not change global data-structures, communicate over global channels nor declare its own ports or timers. However, it may declare its own data variables and execute (serial) algorithms using them. Parallel code is forbidden — although this is somewhat over-restrictive (only `ALT`s should be disallowed!).

The syntax for a <*value.process*> is:

```
<local.declarations.and.abbreviations>
VALOF
  <process>
  RESULT <list.of.expressions>
```

The <*process*> is any *occam* process (subject to the above restrictions). The <*list.of.expressions*> is as in section 11.1.

For example:

```
average := (VAL INT n IS SIZE X:
            REAL32 sum:
            VALOF
              SEQ
                sum := 0.0 (REAL32)
                SEQ i = 0 FOR n
                  sum := sum + x [i]
              RESULT IS sum/n)
```

A <*value.process*> may be used in an expresssion anywhere its resulting <*list.of.expressions*> would be legal. However, its real practical role is to set up the mechanism for an *occam* function.

A `FUNCTION`, like a <*value.process*>, yields a list of data values, occurs only in an expression, causes no side-effects and is deterministic:

```
<list.of.types> FUNCTION <name> (<parameters>)
  <value.process>
:
```

<to be continued>

11.3. Case Process

<coming soon>

11.4. More Local Declarations

<coming soon>

11.5. Special Operators

<coming soon>

11.6. Placing Channels and Variables

<coming soon>

11.7. Placing Processes

<coming soon>