Shared Channels etc.

Peter Welch (p.h.welch@kent.ac.uk)
Computing Laboratory, University of Kent at Canterbury

Co631 (Concurrency)
A Few More Bits of \textit{occam-π}

- **\texttt{SHAREDSHARED}** channels …
- **\texttt{PROTOCOL}** inheritance …
- **\texttt{CASE}** processes …
- Parallel assignment …
- Extended rendezvous …
- Abbreviations and anti-aliasing …
- **\texttt{FUNCTION}**s …
- **\texttt{RECORD}** data types …
- Array slices …
Unshared Channel-Ends

So far, all channels have been strictly *point-to-point* …

Only *one* process may output to it …

And only *one* process may input from it …

clean and simple
Here is a channel whose *writing-end* is *SHARED* …

*Any number* of processes may output to it …

Only *one* process may input from it …

However, only *one* of outputting processes may use it at one time … they form an orderly *(FIFO)* queue for this.
Here is a channel whose *writing-end* is **SHARED** …

```plaintext
SHARED ! CHAN MY.PROTOCOL c:
PAR
  PAR i = 0 FOR n
  smiley (c!)
  server (c?)
```

This allows the writing end to be **SHARED**.
Shared Channel-Ends (*Writers*)

The process at the *reading-end* sees a normal channel …

PROC server (CHAN MY.PROTOCOL in?)

... *normal coding* :

*server* is unaware that the other end of its input channel is *SHARED*.

*server* does not care which process sends it messages.
Shared Channel-Ends (Writers)

The process at the writing-end sees a **SHARED** channel ...

```
PROC smiley (SHARED CHAN MY.PROTOCOL out!)
  ... smiley code body
:
```

**smiley** is aware that its end of the channel is **SHARED**.
Shared Channel-Ends (Writers)

A **shared** channel must be **claimed** before it can be used ...

```plaintext
PROC smiley (SHARED CHAN MY.PROTOCOL out!)
SEQ
... stuff
CLAIM out!
... write to the 'out!' channel
... more stuff
```

Cannot use 'out!' here (unless similarly claimed)
A **Shared** channel must be **claimed** before it can be used …

```
PROC smiley (SHARED CHAN MY.PROTOCOL out!)

SEQ

    ... stuff
    CLAIM out!
    ... write to the 'out!' channel
    ... more stuff
```

This process waits here … until it’s its turn …
Shared Channel-Ends (*Writers*)

A **shared** channel must be **claimed** before it can be used ...

![Diagram with shared channel and server]

PROC smiley (SHARED CHAN MY.PROTOCOL out!)
SEQ
  ... *stuff*
  CLAIM out!
  ... *write to the 'out!' channel*
  ... *more stuff*
  :
Shared Channel-Ends *(Readers)*

Here is a channel whose *reading-end* is SHARED …

Any number of processes may input from it …

Only one process may output to it …

However, only one of inputting processes may use it at one time … they form an orderly *(FIFO)* queue for this.
Here is a channel whose **reading-end** is **SHARED** ...
Shared Channel-Ends (Readers)

The process at the **writing-end** sees a normal channel …

**PROC** generator (CHAN MY.PROTOCOL out!)

... **normal coding**

...  

**generator** is unaware that the other end of its output channel is **SHARED**.

**generator** does not care which process takes its messages.
Shared Channel-Ends *(Readers)*

The process at the *reading-end* sees a *SHARED* channel ...

```
PROC smiley (SHARED CHAN MY.PROTOCOL in?)
  ...  smiley code body
:
```

*smiley* is aware that its end of the channel is *SHARED*.
A **SHARED** channel must be **claimed** before it can be used ...

**PROC** smiley (SHARED CHAN MY.PROTOCOL in?)

**SEQ**

... **stuff**

CLAIM in?

... read from the ‘in?’ channel

... more **stuff**

**Cannot use ‘in?’ here (unless similarly claimed)**
A **shared** channel must be **claimed** before it can be used …

PROC smiley (SHARED CHAN MY.PROTOCOL in?)
SEQ
… stuff
CLAIM in?
… read from the ‘in?’ channel
… more stuff
:

This process waits here … until it’s its turn …
A **shared** channel must be **claimed** before it can be used ...
Shared Channel-Ends (Both)

Here is a channel both of whose ends are **SHARED** …

![Diagram of a shared channel with shared ends.](image)

*Any number* of processes may output to it …

*Any number* of processes may input from it …

However, only **one** outputting process and **one** inputting process may use it at one time … they form an orderly **(FIFO) queue** *at each end.*
Here is a channel both of whose ends are **SHARED** ...
Shared Channel-Ends (Both)

The processes at the writing-end see a **SHARED** channel ...

![Diagram showing processes at writing-end and shared channel]

**PROC** blue.smiley (SHARED CHAN MY.PROTOCOL out!)
... blue.smiley code body

: 

**blue.smiley** is aware that its end of the channel is **SHARED**.

**blue.smiley** will have to **CLAIM** its 'out!' channel to be able to use it.
Shared Channel-Ends *(Both)*

The processes at the **writing-end** see a **SHARED** channel ...

PROC blue.smiley (SHARED CHAN MY.PROTOCOL out!)
...  blue.smiley code body

blue.smiley is unaware of the *sharing* status at the other end.

- blue.smiley must not care which process takes its messages.
Shared Channel-Ends (Both)

The processes at the reading-end see a SHARED channel ...

PROC green.smiley (SHARED CHAN MY.PROTOCOL in?)
  ... green.smiley code body

\begin{itemize}
  \item green.smiley is aware that its end of the channel is SHARED.
  \item green.smiley will have to CLAIM its ‘in?’ channel to be able to use it.
\end{itemize}
Shared Channel-Ends (Both)

The processes at the reading-end see a SHARED channel ...

PROC green.smiley (SHARED CHAN MY.PROTOCOL in?)
  ... green.smiley code body

green.smiley is unaware of the sharing status at the other end.

green.smiley must not care which process sends it messages.
**PROBLEM:** once a sender and receiver process have made their claims, they can do business across the shared channel bundle. Whilst this is happening, all other sender and receiver processes are locked out from the communication resource.

**SOLUTION:** use the shared channel structure just to enable senders and receivers to find each other and pass between them a mobile private channel. Then, let go of the shared channel and transact business over the private connection.
A *sending* process constructs both ends of an *unshared* mobile channel and *claims* the *writing-end* of the shared channel. When successful, it sends the *reading-end* of its mobile channel down the shared channel. This blocks until a *reading* process *claims* its end of the shared channel and inputs that *reading-end* of the mobile.
The **sending** and **reading** processes now exit their **claims** on the shared channel and conduct business over their private connection. Meanwhile, other **senders** and **readers** can use the shared channel similarly and find each other.

Once each **sending** and **reading** pair finish their business, there is a mechanism for the **reader** to return its **reading-end** of the **mobile** channel back to the **sender**, who may then reuse it to send to someone else.

`For info only ...`
A Few More Bits of \texttt{occam}–\(\pi\)

- \textbf{SHARED} channels …
- \textbf{PROTOCOL} inheritance …
- \textbf{CASE} processes …
- Parallel assignment …
- Extended rendezvous …
- Abbreviations and anti-aliasing …
- \textbf{FUNCTION}s …
- \textbf{RECORD} data types …
- Array slices …
Protocol Inheritance (Variant)

A **variant** (or **CASE**) **PROTOCOL** can **extend** previously defined ones:

**PROTOCOL A**

```plaintext
CASE
  red; INT; BYTE::[]BYTE
  green; BYTE; BYTE; INT
  poison
: *
```

**PROTOCOL B**

```plaintext
CASE
  blue; INT::[]REAL64
  poison
: *
```

**PROTOCOL C EXTENDS A, B:**

```plaintext

```
Protocol Inheritance (Variant)

Processes with parameter channels carrying the \( A \) or \( B \) protocols may be plugged into channels carrying \( C \):

**PROTOCOL A**
```
CASE
    red; INT; BYTE::{[]}BYTE
    green; BYTE; BYTE; INT
    poison
```

**PROTOCOL B**
```
CASE
    blue; INT::{[]}REAL64
    poison
```

**PROTOCOL C EXTENDS A, B:**
Protocol Inheritance (Variant)

Processes with parameter channels carrying the **A** or **B** protocols may be plugged into channels carrying **C**:

```plaintext
SHARED ! CHAN C service:
PAR
serve.c (service?)
gen.a (service!)
gen.b (service!)
```
Protocol Inheritance (Variant)

Processes with parameter channels carrying the A or B protocols may be plugged into channels carrying C:

\[\text{PROC } \text{gen.a} \text{ (CHAN A out!) \hspace{1cm} PROC } \text{gen.b} \text{ (CHAN B out!)}\]
\[\begin{align*}
\text{... } & \text{gen.a code body} \hspace{1cm} \text{... } \text{gen.b code body} \\
\text{serve.c} & \end{align*}\]
Protocol Inheritance (Variant)

The extended protocol carries a *merge* of the variants in the protocols it is inheriting.

```
PROTOCOL A
CASE
  red; INT; BYTE::[]BYTE
  green; BYTE; BYTE; INT
: poison

PROTOCOL B
CASE
  blue; INT::[]REAL64
: poison

PROTOCOL C EXTENDS A, B:
```
Protocol Inheritance (Variant)

The extended protocol carries a *merge* of the variants in the protocols it is inheriting. It is as though it were declared:

```
PROTOCOL C
  CASE
    red; INT; BYTE::[]BYTE
    green; BYTE; BYTE; INT
    blue; INT::[]REAL64
    poison
  :
```

*Note:* the above is *not* the same as before … a channel carrying *this* version of the `C` protocol could *not* be plugged into processes expecting `A` or `B` channels.
Protocol Inheritance (Variant)

Rule: protocols being extended together either have no tag names in common or the structures associated with common tags must be identical:

**PROTOCOL A**
```plaintext
CASE
  red; INT; BYTE::[]BYTE
  green; BYTE; BYTE; INT
  poison
: 
```

**PROTOCOL B**
```plaintext
CASE
  blue; INT::[]REAL64
  poison
: 
```

**PROTOCOL C EXTENDS A, B:**

C will compile … compatible variants (poison) from A and B
Protocol Inheritance (Variant)

Rule: protocols being extended together either have no tag names in common or the structures associated with common tags must be identical:

PROTOCOL A
CASE
  red; INT; BYTE::[]BYTE
  green; BYTE; BYTE; INT
  poison; INT
:

PROTOCOL B
CASE
  blue; INT::[]REAL64
  poison; BYTE
:

PROTOCOL C EXTENDS A, B:

C will not compile ... incompatible variants (poison) from A and B
Protocol Inheritance (Variant)

Protocols extending other protocols may also add in their own variants:

```
PROTOCOL C EXTENDS A, B
CASE
    mustard; INT; BYTE::[]BYTE
    aubergine; REAL64; BYTE
:?
```

**Rule:** extra variants so added must have *either* different tag names to any variants being inherited *or* identical structures.
Protocol Inheritance (Variant)

Current implementation restriction: all protocols in an inheritance hierarchy must be declared in the same file.

**PROTOCOL A**
```plaintext
CASE
  red; INT; BYTE::*[]BYTE
  green; BYTE; BYTE; INT
  poison
```

**PROTOCOL B**
```plaintext
CASE
  blue; INT::*[]REAL64
  poison
```

**PROTOCOL C EXTENDS A, B**
```plaintext
CASE
  mustard; INT; BYTE::*[]BYTE
  aubergine; REAL64; BYTE
```

A Few More Bits of \textit{occam-π}

- \textbf{SHARED} channels …
- \textbf{PROTOCOL} inheritance …
- \textbf{CASE} processes …
- Parallel assignment …
- Extended rendezvous …
- Abbreviations and anti-aliasing …
- \textbf{FUNCTION}s …
- \textbf{RECORD} data types …
- Array slices …
Process Structures

There are 6 process constructors ...

SEQ  PAR  ALT  IF  WHILE  CASE

New!
CASE Process

CASE <expression>
<case-list>
<process>
<case-list>
<process>
<case-list>
<process>
<case-list>
<process>
<case-list>
<process>

must be of a discrete type …

BOOL, BYTE, INT, INT16, INT32, INT64

a comma-separated list of compiler-known values from that type …
CASE <expression>
<case-list>
<process>
<case-list>
<process>
<case-list>
<process>
<case-list>
<process>

The <expression> is evaluated.
The first <process> whose <case-list> contains the value of that <expression> is executed.
If no <case-list> contains the value of that <expression>, this process **skips**.
CASE Process

CASE <expression>
  <case-list>
    <process>
  </case-list>
  <case-list>
    <process>
  </case-list>
  <case-list>
    <process>
  </case-list>
  ELSE
    <process>
An optional ELSE <process> may be appended ...

If no <case-list> contains the value of that <expression>, the ELSE <process> is executed.
CASE Process

```java
CASE ch
    'a', 'e', 'i', 'o', 'u'
        ... deal with lower-case vowels
    'A', 'E', 'I', 'O', 'U'
        ... deal with upper-case vowels
    '0', '1', '2', '3', '4'
        ... deal with these digits
    '?', '!', 'h', 'H', '**'
        ... deal with these symbols
ELSE
    ... none of the above
```

Java / C has a similar mechanism – the `switch` statement ...
Java switch Statement

switch (ch) {
    case 'a': case 'e': case 'i': case 'o': case 'u':
        ... deal with lower-case vowels
        break;
    case 'A': case 'E': case 'I': case 'O': case 'U':
        ... deal with upper-case vowels
        break;
    case '0': case '1': case '2': case '3': case '4':
        ... deal with these digits
        break;
    case '?': case '!': case 'h': case 'H': case '*':
        ... deal with these symbols
        break;
    default:
        ... none of the above
}
CASE Process

CASE ch
    'a', 'e', 'i', 'o', 'u'
    ... deal with lower-case vowels
    'A', 'E', 'I', 'O', 'U'
    ... deal with upper-case vowels
    '0', '1', '2', '3', '4'
    ... deal with these digits
    '?', '!', 'h', 'H', '**'
    ... deal with these symbols
ELSE
    ... none of the above

This could, of course, be done with an IF ...

... but it would be more complicated and slower in execution.
CASE Process

IF
(ch = 'a') OR (ch = 'e') OR (ch = 'i') OR
(ch = 'o') OR (ch = 'u')
... deal with lower-case vowels
(ch = 'A') OR (ch = 'E') OR (ch = 'I') OR
(ch = 'O') OR (ch = 'U')
... deal with upper-case vowels
(ch = '0') OR (ch = '1') OR (ch = '2') OR
(ch = '3') OR (ch = '4')
... deal with these digits
(ch = '?') OR (ch = '!') OR (ch = 'h') OR
(ch = 'H') OR (ch = '**')
... deal with these symbols
TRUE
... none of the above

... but it would be more complicated and slower in execution.
| **SHAREDSHARED** | channels … |
| **PROTOCOLPROTOCOL** | inheritance … |
| **CASECASE** | processes … |
| Parallel assignment … |
| Extended rendezvous … |
| Abbreviations and anti-aliasing … |
| **FUNCTIONFUNCTION**s … |
| **RECORDRECORD** data types … |
| Array slices … |
Parallel Assignment

Multiple expressions can be assigned to multiple variables (of compatible types) \textit{in parallel}:

\[
\begin{align*}
a, b, c &:= x, y+1, z-2 \\
\end{align*}
\]

\[
\begin{align*}
\text{REAL32} &\ a\.\ tmp: \\
\text{INT} &\ b\.\ tmp, c\.\ tmp: \\
\text{SEQ} &\ \\
\text{PAR} &\ \\
\ a\.\ tmp &:= x \\
\ b\.\ tmp &:= y+1 \\
\ c\.\ tmp &:= z-2 \\
\text{PAR} &\ \\
\ a &:= a\.\ tmp \\
\ b &:= b\.\ tmp \\
\ c &:= c\.\ tmp
\end{align*}
\]

\textit{First}: the RHS expressions are evaluated \textit{in parallel}.

\textit{Second}: the values are assigned to the target variables \textit{in parallel}. 
Parallel Assignment

Multiple expressions can be assigned to multiple variables (of compatible types) *in parallel*:

\[
\begin{align*}
a, b, c &:= x, y+1, z-2 \\
\end{align*}
\]

\[
\begin{align*}
\text{REAL32 } &a\.tmp:\n\text{INT } &b\.tmp, c\.tmp:\n\text{SEQ} \\
\text{PAR} \\
a\.tmp &:= x \\
b\.tmp &:= y+1 \\
c\.tmp &:= z-2 \\
\text{PAR} \\
a &:= a\.tmp \\
b &:= b\.tmp \\
c &:= c\.tmp
\end{align*}
\]

*Note: parallel usage rules implied by the expanded definition apply to the parallel assignment.*
Parallel Assignment

Swapping variables breaks no *parallel usage rules* and is, therefore, allowed:

\[
\begin{align*}
b, c & := c, b \\
\end{align*}
\]

\[
\begin{align*}
\text{INT } & b.\text{tmp}, c.\text{tmp}: \\
\text{SEQ } & \\
\text{PAR } & \\
& b.\text{tmp} := c \\
& c.\text{tmp} := b \\
\text{PAR } & \\
& b := b.\text{tmp} \\
& c := c.\text{tmp}
\end{align*}
\]

*Note:* parallel assignment is not actually *implemented* in this way. This transformation just defines semantics.
Parallel Assignment

Here’s an example that breaks the \textit{parallel usage rules} and, therefore, does not compile:

\begin{itemize}
\item \texttt{a[i], i := 4.2, 8}
\end{itemize}

\begin{itemize}
\item \texttt{REAL32 a.i.tmp:}
\item \texttt{INT i.tmp:}
\item \texttt{SEQ}
\item \texttt{PAR}
\item \texttt{a.i.tmp := 4.2}
\item \texttt{i.tmp := 8}
\item \texttt{PAR}
\item \texttt{a[i] := a.i.tmp}
\item \texttt{i := i.tmp}
\end{itemize}

\textit{Illegal:} variable ‘\(i\)’ is being changed and observed \textit{in parallel}. 
A Few More Bits of \textit{occam-\pi}

- **SHARED** channels …
- **PROTOCOL** inheritance …
- **CASE** processes …
- Parallel assignment …
- Extended rendezvous …
- Abbreviations and anti-aliasing …
- **FUNCTION**s …
- **RECORD** data types …
- Array slices …
Extended Rendezvous

This is a *convenience* – and it’s free (no impact on run-time).

\[
\text{SEQ} \quad \ldots \\
\quad \ldots \\
\quad \text{in } ?? \times \\
\quad \ldots \text{ rendezvous block} \\
\quad \ldots
\]

The outputting process is unaware of the *extended* nature of the rendezvous.

wait for input; when it arrives, *do not* reschedule the outputting process!

reschedule outputting process only after the rendezvous block has terminated.
Extended Rendezvous

They can be used as **ALT** guards:

```
ALT
  a ? x
    ... react
in ?? x
    ... rendezvous block
    ... react (optional and outside the rendezvous)
tim ? AFTER timeout
    ... react
```
Extended Rendezvous

Here is an informal operational semantics:

\[ c ! 42 \xrightarrow{c} c ?? x \xrightarrow{...} \text{rendezvous block} \]

\[ \equiv \]

\[ \begin{align*}
\text{BOOL any:} & \\
\text{SEQ} & \\
& c ! 42 \\
& c.ack ? any \\
\text{SEQ} & \\
& c ? x \\
& c.ack ! \text{TRUE}
\end{align*} \]

The second version requires an extra channel and for both the sender and receiver processes to be modified.
Extended Rendezvous

Of course, it’s not implemented that way!

- **No new run-time overheads** for normal channel communication.

- Implementation is very lightweight *(approx. 30 cycles)*:
  - *no change* in outputting process code;
  - new *occam Virtual Machine* instructions for “??”.

![Diagram](image)
Extended Rendezvous Tap

Take *any* communication channel …

**Question:** can we *tap* the information flowing through the channel in a way that is not detectable by the existing network?

We may need to do this for data logging (*auditing/de-bugging*) *or* for inserting *network drivers* to implement the channel over a distributed system *or* …
Extended Rendezvous Tap

Take *any* communication channel ... 

**Question:** can we *tap* the information flowing through the channel in a way that is not detectable by the existing network?

**Answer:** insert a process that behaves similarly to an *id* process, but uses an *extended rendezvous* to forward the messages ... *and anything else it fancies (so long as it doesn’t get blocked indefinitely)* ...
Extended Rendezvous Tap

Take *any* communication channel ...

\[
\text{PROC tap (VAL INT id,} \\
\quad \text{CHAN INT in?, out!,} \\
\quad \text{SHARED CHAN LOG log!)}
\]

\[\ldots \text{ tap body} \]

\[
:\quad \text{in} \quad \text{out} \\
\quad \text{tap (id)} \quad \text{log}
\]
**Extended Rendezvous Tap**

Take *any* communication channel …

```plaintext
{{
  {{tap body
    WHILE TRUE
      INT x:
      in ?? x
      PAR
      CLAIM log!
      log ! id; x
      out ! x
  }}
}}
```
Extended Rendezvous Tap

Take any communication channel ...

Note: the channel has been tapped with no change to the sending and receiving processes.

The semantics of a communication between the original processes is unaltered. The sender cannot complete its communication until the receiver takes it ... and vice-versa.
# A Few More Bits of occam-π

| **SHARED** channels … |
| **PROTOCOL** inheritance … |
| **CASE** processes … |
| Parallel assignment … |
| Extended rendezvous … |
| Abbreviations and anti-aliasing … |
| **FUNCTION**s … |
| **RECORD** data types … |
| Array slices … |
Abbreviations and Anti-Aliasing

**Aliasing** means having **different names** for the **same thing**.

**Aliasing** is uncontrolled in most existing languages *(such as Java, C, Pascal, ...)* and gives rise to **semantic complexities** that are underestimated. These complexities are subtle, easy to overlook and cause errors that are hard to find and remove.

**Aliasing** is strictly controlled in **occam-π**. Only **VAL constants** may have different names. Anything else *(variable data, channels, timers, ...)* is only allowed **one name in any one context**. If a **new name** is introduced *(e.g. through parameter passing)*, the **old name** cannot be used within the scope of that new name.

As a result, **occam-π** variables behave in the way we expect variables to behave: *they vary if and only if we vary them*. 😊
Abbreviations and Anti-Aliasing

Reference Abbreviation:

<specifier> <new-name> IS <old-name>:

<data-type>
CHAN <protocol>
TIMER
...

<process>

<old-name> is not allowed in here

<new-name>

scope of

<new-name>
Abbreviations and Anti-Aliasing

Reference Abbreviation:

Any variables *(e.g. array indices)* used in determining *<old-name>...*

*<specifier> <new-name> IS <old-name>:

*<process>*

*<old-name>* is not allowed in here

*<old-name>* are frozen in the scope of *<new-name>*
Abbreviations and Anti-Aliasing

Reference Abbreviation:

Example

\[ [200][100] \text{REAL64 } x \]

\[ \text{INT } n \]

\[ \text{INT } i \]

\[ \text{CHAN MY.PROTOCOL } c! \]

\[ \text{INT result IS } n:\]
\[ \text{REAL64}[] \text{ row.i IS } x[i]: \]
\[ \text{CHAN MY.PROTOCOL } \text{out! IS } c!: \]

Cannot refer to \( n, x[i] \) or \( c! \) in here.

Can refer to \( i \) in here, but can’t change it.
Abbreviations and Anti-Aliasing

Reference Abbreviation:
Example

\[ [200][100]\text{REAL64 } x \]

\text{CHAN MY.PROTOCOL c!}

\text{INT n}

\text{INT i}

\text{INT j}

\begin{align*}
\text{INT result IS n:} \\
\text{REAL64[]} \text{ row.i IS x[i]:} \\
\text{CHAN MY.PROTOCOL out! IS c!:}
\end{align*}

Can refer to \( x[j] \) here … but only if \( i \neq j \). If the compiler doesn’t know, a run-time check will be made.
Abbreviations and Anti-Aliasing

Value Abbreviation:

\[
\text{VAL} \quad \text{<data-type>} \quad \text{<name>} \quad \text{IS} \quad \text{<expression>}:
\]

<expression> must match the <data-type>

<process>

<name> cannot be changed in here

scope of <name>
Abbreviations and Anti-Aliasing

Value Abbreviation:

```
VAL <data-type> <name> IS <expression>:
```

Any variables used in `<expression>` ...

... are frozen in the scope of `<name>`

`<name>` cannot be changed in here
Abbreviations and Anti-Aliasing

Value Abbreviation:

Example

**REAL64** \(a\)

**REAL64** \(b\)

**INT** \(i\)

\([200][100]** **REAL64** \(x\)

---

**VAL** \(\text{REAL64 hypotenuse IS SQRT } ((a^a) + (b^b)):\)

**VAL** \(\text{REAL64[ ] row.}i \ IS x[i]:\)

**VAL** \(\text{INT } n \ IS SIZE row.}i:\)

\(<\text{process}>\)

---

Cannot change **hypotenuse**, **row.}i** or **n** in here.

Also, cannot change **a**, **b**, **i** or **x[i]** in here.
Abbreviations and Anti-Aliasing

Careful use of abbreviations can clarify code and increase efficiency.

Here’s simple code for adding up the elements of a 1-D array:

```
SEQ
sum := 0
SEQ i = 0 FOR SIZE a
    sum := sum + a[i]
```
Abbreviations and Anti-Aliasing

Now, let’s add up the rows of a 2-D array:

```
<table>
<thead>
<tr>
<th>a</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(m-1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>(m-1)</td>
</tr>
</tbody>
</table>
```

SEQ row = 0 FOR SIZE a

SEQ
def sum[row] := 0
SEQ col = 0 FOR SIZE a[row]
Abbreviations and Anti-Aliasing

This code contains some wasteful re-computations:

SEQ row = 0 FOR SIZE a
SEQ
  sum[\text{row}] := 0
SEQ col = 0 FOR SIZE a[\text{row}]
  sum[\text{row}] := sum[\text{row}] + a[\text{row}][\text{col}]

For each ‘\text{row}’, the address of ‘\text{sum[\text{row}]}’ is calculated \((2n+1)\) times – where ‘\text{n}’ is the size of the ‘\text{row}’.

For each ‘\text{row}’, the address of ‘\text{a[\text{row}]}’ is calculated \((n+1)\) times – where ‘\text{n}’ is the size of the ‘\text{row}’.

With abbreviations, the addresses of ‘\text{sum[\text{row}]}’ and ‘\text{a[\text{row}]}’ need only be calculated \textit{once} for each ‘\text{\text{row}}’ … a saving of \((3n^2m)\) \textit{array index computations}, over ‘\text{m}’ rows.  😊😊😊
Abbreviations and Anti-Aliasing

We just abbreviate ‘\texttt{sum[row]}’ and ‘\texttt{a[row]}’:

\begin{verbatim}
SEQ row = 0 FOR SIZE a
    INT sum.row IS sum[row]:
    VAL []INT a.row IS a[row]:

    SEQ
        sum.row := 0
        SEQ col = 0 FOR SIZE a.row
            sum.row := sum.row + a.row[col]
\end{verbatim}

The neat thing is that, following the abbreviations, the inner loop code is exactly the same (bar variable names) as the original summation code for the 1-D loop:

\begin{verbatim}
SEQ
    sum := 0
    SEQ i = 0 FOR SIZE a
        sum := sum + a[i]
\end{verbatim}
Parameters and Abbreviations

An \texttt{occam-\pi} \texttt{PROC} call is formally defined as the \textit{in-line replacement} of the invocation with the body of the \texttt{PROC}, proceeded by a sequence of abbreviations associating the formal parameters (\texttt{<new-names>}) with the actual arguments (\texttt{<old-names>} or \texttt{<expressions>}) from the call.

Consider:

\begin{verbatim}
PROC foo (VAL INT id, INT a, b, REAL64[] row,
   CHAN MY.PROTOCOL out!)
   ...
  body of foo (using id, a, b, row, out!)

\end{verbatim}
Parameters and Abbreviations

PROC foo (VAL INT id, INT a, b, REAL64[] row, CHAN MY.PROTOCOL out!)
    ... body of foo (using id, a, b, row, out!):

Now consider an invocation of foo:

    foo (i+1, n, m, x[i], c!)

This is formally defined to be:

    VAL INT id IS i+1:
    INT a IS n:
    INT b IS m:
    REAL64[] row IS x[i]:
    CHAN MY.PROTOCOL out! IS c!:
    ... body of foo (using id, a, b, row, out!)
Parameters and Abbreviations

PROC foo (VAL INT id, INT a, b, REAL64[] row,
CHAN MY.PROTOCOL out!)

... body of foo (using id, a, b, row, out!)

The point is that the **anti-aliasing** rules carry over (from abbreviations) to parameter passing …
Parameters and Abbreviations

PROC foo (VAL INT id, INT a, b, REAL64[] row, 
CHAN MY.PROTOCOL out!)

... body of foo (using id, a, b, row, out!)

The following invocation is illegal:

foo (i+1, n, n, x[i], c!)

This is formally defined to be:

VAL INT id IS i+1:
INT a IS n:
INT b IS n:
REAL64[] row IS x[i]:
CHAN MY.PROTOCOL out! IS c!:

This attempts to set up a and b as aliases of n.

We are not allowed to mention n here.
Parameters and Abbreviations

PROC foo (VAL INT id, INT a, b, REAL64[] row, 
CHAN MY.PROTOCOL out!)

...  body of foo (using id, a, b, row, out!)

The following invocation is illegal:

foo (i+1, n, n, x[i], c!)

This is formally defined to be:

VAL  INT  id  IS  i+1:
  INT  a  IS  n:
  INT  b  IS  n:
REAL64[]  row  IS  x[i]:
CHAN  MY.PROTOCOL  out!  IS  c!:

...  body of foo (using id, a, b, row, out!)

Therefore, this does not compile.

We are not allowed to mention n here.
Anti-Aliasing

Recall, \texttt{occam-\pi} variables behave in the way we expect variables to behave: \textit{they vary if and only if we vary them}.

Consider the fragment of code:

\begin{verbatim}
SEQ
  a := a + b
  a := a - b
\end{verbatim}

Assume the arithmetic does not overflow.

Everything we feel about algebra, variables, assignment and sequencing tells us: \textit{the above code changes nothing}.

For all languages providing algebra, variables, assignment and sequencing – apart (currently) from \texttt{occam-\pi} – that intuition will lead us astray.
Anti-Aliasing

There is a potential semantic singularity below:

```
SEQ
  a := a + b
  a := a - b
```

Assume the arithmetic does not overflow.

The above code changes nothing ... only if \( a \) and \( b \) reference different numbers.

If \( a \) and \( b \) reference the same number, they would both end up with zero! The value of \( b \) would vary without it being explicitly varied.
Anti-Aliasing

There is a potential semantic singularity below:

```
SEQ
a := a + b
a := a - b
```

Assume the arithmetic does not overflow.

The above code changes nothing … only if \( a \) and \( b \) reference different numbers.

If \( a \) and \( b \) reference the same number, they would both end up with zero! The value of \( b \) would vary without it being explicitly varied.

A complex and horrid semantics …
Anti-Aliasing

What You See Is What You Get (WYSIWYG)

That kind of nonsense does not happen in occam-\(\pi\):

```seq
a := a + b
a := a - b
```

Assume the arithmetic does not overflow.

The above code changes nothing ... we know that \(a\) and \(b\) reference different numbers.

The anti-aliasing rules mean that different variables in the same context must refer to different items.
Aliasing and Java etc.

What You See Is *Not* What You Get *(WYSINWYG)*

Java has no aliasing problems with its primitive types … but aliasing is part of the culture of ‘Object Orientation’ … it is endemic to ‘OO’ … *we have to work very hard to control it.*

Consider:

```
    a.plus (b);
    a.minus (b);
```

where `a` and `b` are object variables of the same class … with some private field holding an integer whose value is updated by the `plus` and `minus` methods in the obvious way …

Assume the arithmetic does not overflow.
Aliasing and Java etc.

What You See Is Not What You Get (WYSINWYG)

class Thing {
    private integer sum = 0;
    public void plus (Thing t) {sum = sum + t.sum;}
    public void minus (Thing t) {sum = sum - t.sum;}
    ... other methods
}

If Thing variables a and b reference the same object, they would end up holding zero in their sum field! The value of b varies without it being (explicitly) updated.
What You See Is Not What You Get (WYSINWYG)

If Thing variables \texttt{a} and \texttt{b} reference the same object, they would end up holding zero in their sum field! The value of \texttt{b} varies without it being (explicitly) updated.

\begin{verbatim}
    a.plus (b);
    a.minus (b);
\end{verbatim}
Aliasing and Java etc.

What You See Is *Not* What You Get *(WYSINWYG)*

If **Thing** variables *a* and *b* reference the same object, *they* would end up holding zero in their **sum field**! The value of *b* **varies** without it being (explicitly) updated.

This is not an uncommon piece of coding … we often write:

... set up object *a*
... use *a* for something
... restore *a* to its previous state

**BUT IT’S BEEN ZEROED!!!**
A Few More Bits of occam-π

- **shared** channels …
- **protocol** inheritance …
- **case** processes …
- Parallel assignment …
- Extended rendezvous …
- Abbreviations and anti-aliasing …
- **function**s …
- **record** data types …
- Array slices …
VALOF Expressions

\[
\begin{align*}
\langle \text{process}\rangle \\
\langle \text{list-of-expressions}\rangle \\
\end{align*}
\]

This allows us to declare variables in the middle of expressions and perform calculations \textit{(serial logic only)}. If the result list has more than one item, this can only be the \textit{Right-Hand-Side} of a parallel assignment.
VALOF Expressions

REAL64 total

[1000]REAL64 x

total := total +

(REAL64 sum:
  VALOF
  SEQ
  sum := 0
  SEQ i = 0 FOR SIZE x
  sum := sum + x[i]
  RESULT sum
)
VALOF Expressions

\[
\begin{align*}
&\text{BYTE } a, \text{ REAL32 } b, \text{ BYTE } c : = \text{ (BYTE } \text{ ch, sh: REAL32 } z: \text{ VALOF)} \\
&\text{RESULT } \text{ ch, z, sh }
\end{align*}
\]
Functions

\[
\langle \text{type-list}\rangle \text{ FUNCTION } \langle \text{id}\rangle \ (\langle \text{params}\rangle)
\]

\[
\langle \text{local-declarations}\rangle
\]

\[
\text{VALOF}
\]

\[
\langle \text{process}\rangle
\]

\[
\text{RESULT } \langle \text{list-of-expressions}\rangle
\]

must match the \[
\langle \text{type-list}\rangle
\]

The \[
\langle \text{params}\rangle
\] may only be VAL data types (no reference data, channels, …).

Functions are deterministic and side-effect free (i.e. its body may not assign to global variables, communicate on global channels, use timers or engage in any internal concurrency using ALT or SHARED channels.)
Short Functions

\[
\text{\texttt{\textless type.list\texttt{> FUNCTION \texttt{id} (\texttt{\textless params\texttt{>}) IS}}}
\]

\[
\text{\texttt{\textless list-of-expressions\texttt{>}}}
\]

\[
\text{\texttt{VALUE}}
\]

\[
\text{\texttt{LIB}}
\]

\[
\text{\texttt{MAX}}
\]

\[
\text{\texttt{MIN}}
\]

\[
\text{\texttt{FOR EXAMPLE ...}}
\]

\[
\text{\texttt{BOOL FUNCTION capital (VAL BYTE ch) IS}}
\]

\[
\text{\texttt{('A' \leq \ ch) AND (ch \leq 'Z')}:}}
\]
A Few More Bits of occam-$\pi$

- **SHARED** channels …
- **PROTOCOL** inheritance …
- **CASE** processes …
- Parallel assignment …
- Extended rendezvous …
- Abbreviations and anti-aliasing …
- **FUNCTION**s …
- **RECORD** data types …
- Array slices …
occam-$\pi$ Data Types

**Revision:**

*occam-$\pi$* has a set of *primitive* types:

- BOOL, BYTE, INT, INT16, INT32, INT64, REAL32, REAL64

*occam-$\pi$* has *fixed-size anonymous array* types:

- \([n] \langle\text{type}\rangle\)

where \(n\) is a *compiler-known* INT value and \(<\text{type}\rangle\) is a *compiler-known* type (which could itself be an *array* type).

**New:**

*occam-$\pi$* allows new *named* types to be declared.
**occam-π Data Types**

**Records:**

An **array** type groups together elements of the **same** type. A **record** type groups together elements of **different** types:

```plaintext
DATA TYPE FOO
RECORD
    INT size, weight:
    BYTE colour:
    REAL64 frequency:
    [10]BYTE name:
```

This gives a record with 5 **named fields**: two **INT** ones, one **BYTE**, one **REAL64** and one **BYTE** array (e.g. a string).
**occam-π Data Types**

**Records:**

Now, we can declare variables of this new type:

```
FOO x, y, z:
[42]FOO database:
```

To access individual fields of a record, the notation is like array indexing:

```seq
x[size] := 42
y[weight] := 77
z[name] := "Susan"
z[size] := x[size]
y[name] := z[name]
```
occam-$\pi$ Data Types

Now, we can declare variables of this new type:

\begin{verbatim}
FOO x, y, z:
[42]FOO database:
\end{verbatim}

Record literals let us assign all fields at once:

\begin{verbatim}
x := [42, 77, green, 99.7158214, "Josephson "]
\end{verbatim}

where, perhaps:

\begin{verbatim}
VAL BYTE green IS 6:
\end{verbatim}
**occam-\(\pi\) Data Types**

**Records:**

*Record* data types are *first class* types. We can assign them to each other or send them down appropriately typed channels:

\[
\text{SEQ } x, y : \text{FOO} \\
\begin{align*}
  x & := [42, 77, \text{green}, 99.7158214, "Josephson"] \\
  \ldots & \text{ stuff} \\
  y & := x
\end{align*}
\]

*Note:* in *Java*, assignment between object variables just copies the reference. The source and target variables end up referring to the *same* object.
occam-$\pi$ Data Types

**Records:**

*Record* data types are *first class* types. We can assign them to each other or send them down appropriately typed channels:

```
FOO x, y:
SEQ
  x := [42, 77, green, 99.7158214, "Josephson "]
  ... stuff
  y := x
```

All the data in *x* is copied into *y*.

*Note:* in *occam*-\(\pi\), assignment between variables copies the data. The source and target variables end up referring to *different* pieces of data.
**occam-π Data Types**

**Records:**

*Record* data types are *first class*. We can assign them to each other or send them to properly typed channels:

```plaintext
FOO  x, y:
SEQ  x := [
    42, 77, green, 9.71582149, "Josephson"
  ]... stuff...
   yy := x
```

*All the data in x is copied into y.*

**Note:** in *occam-π*, data may be declared *MOBILE*. For such data, assignment (and communication) moves the data from the source to the target – leaving the source variable referring to no data. [This is for information only – not part of this course.]

**Note:** in *occam-π*, assignment between variables copies the data. The source and target variables end up referring to different pieces of data.
Records:

Record data types are *first class* types. We can assign them to each other or send them down appropriately typed channels:

```plaintext
CHAN FOO c:
  PAR
    R0 (c!)
    R1 (c?)
```

R0 \rightarrow c \rightarrow R1
Records:

*Record* data types are *first class* types. We can assign them to each other or send them down appropriately typed channels:

```
PROC R0 (CHAN FOO out!)
  FOO x:
  SEQ
    ... set up x
    out ! x
    ... more stuff
    out ! [21, 72, blue, 3.142, "Junction " ]
:  
```

**occam-π Data Types**
**occam-π Data Types**

*Records:*  

*Record* data types are *first class* types. We can assign them to each other or send them down appropriately typed channels:

```occam
PROC R1 (CHAN FOO in?)
    FOO x, y:
    SEQ
        in ? x
        ... stuff
        in ? y
        ... more stuff
    :
```

occam-$\pi$ Data Types

**Renamed Types:**

We can just define a new type to be implemented by an existing type:

- `DATA TYPE COLOUR IS BYTE:`
- `DATA TYPE MATRIX IS [20][30]REAL64:`
- `DATA TYPE BAR IS FOO:`

Now, `COLOUR`, `MATRIX` and `BAR` are new types, different to their underlying `BYTE`, `[20][30]REAL64` and `FOO` types.

*occam-$\pi$* enforces strong typing. So, `COLOUR` and `BYTE` variables are not assignment compatible. Also, a `COLOUR` variable cannot be the target of an input from a `CHAN BYTE` (or vice-versa).
occam-π Data Types

Example:

DATA TYPE COLOUR IS BYTE:
BYTE b:
COLOUR c:
SEQ
...  stuff
b := c   -- illegal: will not compile
...  more stuff
...  stuff
b := c   -- illegal: will not compile

User re-named data types can give extra security against careless errors.

occam-π enforces strong typing. So, COLOUR and BYTE variables are not assignment compatible. Also, a COLOUR variable cannot be the target of an input from a CHAN BYTE (or vice-versa).
**occam-π Data Types**

**Example:**

```
PROC foo (CHAN COLOUR colour.in?, colour.out!,
          CHAN BYTE byte.in?, byte.out!)

BYTE b:
COLOUR c:
SEQ
  colour.in ? b -- illegal: will not compile
  colour.out ! b -- illegal: will not compile
  byte.in ? c  -- illegal: will not compile
  byte.out ! c  -- illegal: will not compile
:
```

*occam-π* enforces **strong typing**. So, *COLOUR* and *BYTE* variables are not assignment compatible. Also, a *COLOUR* variable cannot be the target of an input from a *CHAN BYTE* (or vice-versa).
Example:

```
PROC foo (CHAN COLOUR colour.in?, colour.out!,
           CHAN BYTE byte.in?, byte.out!)

BYTE b:
COLOUR c:
SEQ
   colour.in ? c    -- legal
   colour.out ! c   -- legal
   byte.in ? b      -- legal
   byte.out ! b     -- legal
:
```

`occam-π` enforces **strong typing**. So, `COLOUR` and `BYTE` variables are not assignment compatible. Also, a `COLOUR` variable cannot be the target of an input from a `CHAN BYTE` (or vice-versa).

User re-named data types can give extra security against careless errors.
Data types \texttt{FOO}, \texttt{BAR} and \texttt{WIPPY} have the same \texttt{structure} but are not \texttt{equivalent}.
**occam-\(\pi\) Data Types**

*Type Equivalence:*

\(\text{occam-}\pi\) types are *equivalent* if and only if they have the same name.

**DATA TYPE BAR IS FOO:**

```
DATA TYPE FOO
  RECORD
    INT size, weight:
    BYTE colour:
    REAL64 frequency:
    [10]BYTE name:
```

```
DATA TYPE WIPPY
  RECORD
    INT size, weight:
    BYTE colour:
    REAL64 frequency:
    [10]BYTE name:
```

\(\text{FOO, BAR and WIPPY}\) variables may not be directly *assigned* to each other – but their values may be *cast*. 
Type Equivalence:

occam-π types are equivalent if and only if they have the same name.

```plaintext
FOO f:
WIPPY w:
SEQ
  ... set up f
w := f        -- illegal: will not compile
  ... more stuff
w := WIPPY f    -- legal
```

FOO, BAR and WIPPY variables may not be directly assigned to each other – but their values may be cast.
**occam-π** Data Types

**Type Equivalence:**

**occam-π** types are *equivalent* if and only if they have the same name.

```
DATA TYPE MATRIX IS [20][30]REAL64:

MATRIX m:
[20][30]REAL64 x:
SEQ
  ... set up x
m := x          -- illegal: will not compile
  ... more stuff
m := MATRIX x   -- legal
```

**MATRIX** and **[20][30]REAL64** variables may not be directly assigned to each other – but their values may be **cast**.
**occam-π Data Types**

**Type Equivalence:**

**occam-π** types are *equivalent* if and only if they have the same name.

Array types are *anonymous* – but any particular array type has an implicit (hidden) name that is *the same* for all occurrences of that type.

So, [20] [30] REAL64 variables are always *assignable* to each other – wherever they happen to have been declared.
Operator Inheritance:

All arithmetic and logical operators on *primitive* types are *inherited* by types *renaming* them.

**DATA TYPE COLOUR IS BYTE:**

```plaintext
COLOUR red, green, yellow:
SEQ
... set up red and green
yellow := read \ green
... stuff
```
Operator Inheritance:

All indexing and size operations on array types are inherited by types renaming them.

DATA TYPE MATRIX IS [20][30]REAL64:

MATRIX m:
SEQ
  SEQ i = 0 FOR SIZE m
    SEQ j = 0 FOR SIZE m[i]
      m[j][i] := some.real64
      ... stuff
occam-π Data Types

Operator Inheritance:

All field indexing operations on record types are inherited by types renaming them.

DATA TYPE FOO
RECORD
  INT size, weight:
  BYTE colour:
  REAL64 frequency:
[10]BYTE name:
:
DATA TYPE BAR IS FOO:

BAR b:
SEQ
  b[size] := 42
  b[weight] := 77
  b[colour] := yellow
  ... stuff
A Few More Bits of \textit{occam-\(\pi\)}

\begin{itemize}
\item \textbf{SHARED} channels …
\item \textbf{PROTOCOL} inheritance …
\item \textbf{CASE} processes …
\item Parallel assignment …
\item Extended rendezvous …
\item Abbreviations and anti-aliasing …
\item \textbf{FUNCTION}s …
\item \textbf{RECORD} data types …
\item Array slices …
\end{itemize}
Array Slices

Let \( a \) be an array. Then, the expression:

\[
[a \ FROM \ start \ FOR \ n]
\]

represents the \textit{slice} of the array \( a \) from element \( a[\text{start}] \)
through \( a[\text{start} + (n - 1)] \) inclusive. Also:

\[
[a \ FOR \ n]
\]

represents the \textit{slice} consisting of the first \( n \) elements. Also:

\[
[a \ FROM \ start]
\]

represents the \textit{slice} from element \( a[\text{start}] \) to its end.

The defined \textit{slices} must lie within the bounds of the array.
Array Slices

\[[a \text{ FROM 6 FOR 5}]\]
Array Slices

\[ [a \text{ FOR } 12] \]
Array Slices

\[[a \text{ FROM } 8]\]

\[\begin{array}{cccccc}
14 & 13 & 12 & 11 & 10 & 9 \\
8 & 7 & 6 & 5 & 4 & 3 \\
2 & 1 & 0 & max-1 & max-2
\end{array}\]

\(a\)
Array Slices

An array slice may be the source or target of assignment:

\[ [a \text{ FROM } i \text{ FOR } n] := [b \text{ FROM } j \text{ FOR } n] \]

The slice sizes must be the same.

\[ [a \text{ FROM } i \text{ FOR } n] := [a \text{ FROM } j \text{ FOR } n] \]

The slices must not overlap.
Array Slices

An array slice may be the source or target of communication:

\[
\text{out} ! \ [b \ \text{FROM} \ j \ \text{FOR} \ n]
\]

The channel must carry \([n]\) arrays …

\[
\text{in} ? \ [a \ \text{FROM} \ i \ \text{FOR} \ n]
\]

… where \(n\) is a compiler known value.
Array Slices

More flexible (and usual) would be a *counted array* protocol:

$$\text{out} ! \quad n::[b \text{ FROM } j]$$

Output \(n\) elements from \(b[j] \ldots\)

$$\text{in} ? \quad m::[a \text{ FROM } i]$$

Input \(m\) elements starting at \(a[i] \ldots\)