Towards a New Language for Concurrent Programming

CPA-2011 Fringe Session

Fred Barnes
School of Computing, University of Kent, Canterbury

F.R.M.Barnes@kent.ac.uk

http://www.cs.kent.ac.uk/~frmb/

Programming Languages and Systems

Complex Systems Modelling and Simulation Infrastructure

University of Kent | Computing

Background

- We've been knocking around ideas about a new occam for some time..
- Some issues with occam and occam-pi as they currently exist:
  - perceived as an "old" language (or even dead!)
  - upper-case keywords went out of fashion with BASIC
  - strict indentation annoys some
- Occam-pi (as it stands) is essentially a "bolt-on" to occam
  - language is a little inconsistent or clunky in places
  - compiler breaks down easily (old code-base)
- If we're reinventing compilers, might as well reinvent the language whilst we're at it...

Background

- We've been knocking around ideas about a new occam for some time..
- Some issues with occam and occam-pi as they currently exist:
  - perceived as an "old" language (or even dead!)
  - upper-case keywords went out of fashion with BASIC
  - strict indentation annoys some
- Occam-pi (as it stands) is essentially a "bolt-on" to occam
  - language is a little inconsistent or clunky in places
  - compiler breaks down easily (old code-base)
- If we're reinventing compilers, might as well reinvent the language whilst we're at it...

Background

- We've been knocking around ideas about a new occam for some time..
- Some issues with occam and occam-pi as they currently exist:
  - perceived as an "old" language (or even dead!)
  - upper-case keywords went out of fashion with BASIC
  - strict indentation annoys some
- Occam-pi (as it stands) is essentially a "bolt-on" to occam
  - language is a little inconsistent or clunky in places
  - compiler breaks down easily (old code-base)
- If we're reinventing compilers, might as well reinvent the language whilst we're at it...

Background

- We've been knocking around ideas about a new occam for some time..
- Some issues with occam and occam-pi as they currently exist:
  - perceived as an "old" language (or even dead!)
  - upper-case keywords went out of fashion with BASIC
  - strict indentation annoys some
- Occam-pi (as it stands) is essentially a "bolt-on" to occam
  - language is a little inconsistent or clunky in places
  - compiler breaks down easily (old code-base)
- If we're reinventing compilers, might as well reinvent the language whilst we're at it...

Guppy

- Introducing **Guppy**
  - deliberately not called ‘occam’
  - ... although we’re going to use all the best bits :-)
- Still looking for a decent logo ...

What We Need ...

- Preserving the useful features of occam/occam-pi:
  - embodiment of CSP based concurrency (though may not restrict to that alone) in the language itself
  - strict parallel usage checks: zero aliasing
- Preserving the fast execution of the resulting code:
  - no heavy run-time checks (e.g. expensive run-time typing, complex garbage collection)
  - using existing CCSP
- Targetable at just about any architecture in existence:
  - by compiling (ultimately) to LLVM (low-level virtual machine)

What We Need ...

- Preserving the useful features of occam/occam-pi:
  - embodiment of CSP based concurrency (though may not restrict to that alone) in the language itself
  - strict parallel usage checks: zero aliasing
- Preserving the fast execution of the resulting code:
  - no heavy run-time checks (e.g. expensive run-time typing, complex garbage collection)
  - using existing CCSP
- Targetable at just about any architecture in existence:
  - by compiling (ultimately) to LLVM (low-level virtual machine)

What We Need ...

- Preserving the useful features of occam/occam-pi:
  - embodiment of CSP based concurrency (though may not restrict to that alone) in the language itself
  - strict parallel usage checks: zero aliasing
- Preserving the fast execution of the resulting code:
  - no heavy run-time checks (e.g. expensive run-time typing, complex garbage collection)
  - using existing CCSP
- Targetable at just about any architecture in existence:
  - by compiling (ultimately) to LLVM (low-level virtual machine)

What We Would Like ...

- A language that other people would be happy to (and may even want to) use:
  - successes of Python and Go suggest indentation-based layout and concurrency are not distasteful
- Rapid development – nothing overly cumbersome to program with respect to other languages:
  - need some genericity/flexibility in the type system
  - automatic '`SEQ`' behaviour (static checks can spot likely errors)
  - may need to sacrifice some of the purity of occam to make this work..
- Automatic mobility (largely a compiler thing), with a couple of language hints thrown in to help the compiler when automatic static analysis gets too complex (or wrong).
- A proper '`string`' type with UTF-8 support (32-bit '`char`' probably).

What We Would Like ...

- A language that other people would be happy to (and may even want to) use:
  - successes of Python and Go suggest indentation-based layout and concurrency are not distasteful
- Rapid development – nothing overly cumbersome to program with respect to other languages:
  - need some genericity/flexibility in the type system
  - automatic 'SEQ' behaviour (static checks can spot likely errors)
  - may need to sacrifice some of the purity of occam to make this work..
- Automatic mobility (largely a compiler thing), with a couple of language hints thrown in to help the compiler when automatic static analysis gets too complex (or wrong).
- A proper 'string' type with UTF-8 support (32-bit 'char' probably).

What We Would Like ...

- A language that other people would be happy to (and may even want to) use:
  - successes of Python and Go suggest indentation-based layout and concurrency are not distasteful
- Rapid development – nothing overly cumbersome to program with respect to other languages:
  - need some genericity/flexibility in the type system
  - automatic '`SEQ`' behaviour (static checks can spot likely errors)
  - may need to sacrifice some of the purity of occam to make this work..
- Automatic mobility (largely a compiler thing), with a couple of language hints thrown in to help the compiler when automatic static analysis gets too complex (or wrong).
- A proper '`string`' type with UTF-8 support (32-bit '`char`' probably).

What We Would Like ...

- A language that other people would be happy to (and may even want to) use:
  - successes of Python and Go suggest indentation-based layout and concurrency are not distasteful
- Rapid development – nothing overly cumbersome to program with respect to other languages:
  - need some genericity/flexibility in the type system
  - automatic '`SEQ`' behaviour (static checks can spot likely errors)
  - may need to sacrifice some of the purity of occam to make this work..
- Automatic mobility (largely a compiler thing), with a couple of language hints thrown in to help the compiler when automatic static analysis gets too complex (or wrong).
- A proper '`string`' type with UTF-8 support (32-bit '`char`' probably).

Type System

- Usual primitive types:

```
int x                   # simple signed integer
uint14 y                # 14-bit unsigned integer
bool z                  # boolean
real64 f                # floating-point
string s                # string type
char c                  # unicode character
byte b                  # unsigned 8-bit
```

- Structured (and optionally parameterised) types:

- Named types:

Type System

- Usual primitive types:

```
int x                        # simple signed integer
uint14 y                     # 14-bit unsigned integer
bool z                       # boolean
real64 f                     # floating-point
string s                     # string type
char c                       # unicode character
byte b                       # unsigned 8-bit
```

- Structured (and optionally parameterised) types:

```
define type iCoord
  int x, y

iCoord p, o = [0,0]
```

- Named types:

Type System

- Usual primitive types:

```
int x                        # simple signed integer
uint14 y                     # 14-bit unsigned integer
bool z                       # boolean
real64 f                     # floating-point
string s                     # string type
char c                       # unicode character
byte b                       # unsigned 8-bit
```

- Structured (and optionally parameterised) types:

```
define type iCoord
  int x, y

iCoord p, o = [0,0]
```

```
define type Coord (T)
  T x, y

Coord(int) p, o = [0,0]
```

- Named types:

Type System

- Usual primitive types:

```
int x                       # simple signed integer
uint14 y                    # 14-bit unsigned integer
bool z                      # boolean
real64 f                    # floating-point
string s                    # string type
char c                      # unicode character
byte b                      # unsigned 8-bit
```

- Structured (and optionally parameterised) types:

```
define type iCoord
  int x, y

iCoord p, o = [0,0]
```

```
define type Coord (T)
  T x, y

Coord(int) p, o = [0,0]
```

- Named types:

```
define type NanoTime is uint128
```

Channels and Protocols

- Channels are explicitly typed with a specific protocol (as they are in occam), and sometimes with a direction
  - can be a 'null' protocol (what 'SIGNAL' is in occam-pi, more or less).
- First-class types in the language, so can be used as protocols themselves to define things like **channel mobility**.
- Borrow Adam's two-way protocols for defining complex communication patterns (via state machines):
  - related to the idea of **session types**

Channels and Protocols

- Channels are explicitly typed with a specific protocol (as they are in occam), and sometimes with a direction
  - can be a 'null' protocol (what 'SIGNAL' is in occam-pi, more or less).
- First-class types in the language, so can be used as protocols themselves to define things like **channel mobility**.

```
chan?(chan!(int))
chan!(Link)
```

- Borrow Adam's two-way protocols for defining complex communication patterns (via state machines):
  - related to the idea of **session types**

Channels and Protocols

- Channels are explicitly typed with a specific protocol (as they are in occam), and sometimes with a direction
  - can be a 'null' protocol (what 'SIGNAL' is in occam-pi, more or less).
- First-class types in the language, so can be used as protocols themselves to define things like **channel mobility**.

```
chan?(chan!(int))
chan!(Link)
```

- Borrow Adam's two-way protocols for defining complex communication patterns (via state machines):
  - related to the idea of **session types**

```
define protocol Link
  subprotocol State1
    case
      ! start; int
        State2
  subprotocol State2
    case
      ? starting
        State3
      ? failed; int
        State1
  # more states ...
  State1
```

Tuples and Abstract Types

- Anonymous structure (tuple) types (allowed generally as L-values):

```
chan({int,char}) c
par
  c ! {42,'x'}
  c ? {x,y}
```

- Abstract types, which provide an equivalent of a union and allow for recursive data structures (without having to abuse the forward-scoping rules):

- Must supply a '`default`' variant that is used for initialisation.

Tuples and Abstract Types

- Anonymous structure (tuple) types (allowed generally as L-values):

```
chan({int,char}) c
par
  c ! {42,'x'}
  c ? {x,y}
```

- Abstract types, which provide an equivalent of a union and allow for recursive data structures (without having to abuse the forward-scoping rules):

```
define type Tree
define type Leaf is Tree
  int value
define type SubTree is Tree
  Tree left, right
define type Empty is default Tree
```

- Must supply a '`default`' variant that is used for initialisation.

Tuples and Abstract Types

- Anonymous structure (tuple) types (allowed generally as L-values):

```
chan({int,char}) c
par
  c ! {42,'x'}
  c ? {x,y}
```

- Abstract types, which provide an equivalent of a union and allow for recursive data structures (without having to abuse the forward-scoping rules):

```
define type Tree
define type Leaf is Tree
  int value
define type SubTree is Tree
  Tree left, right
define type Empty is default Tree

Tree t
case t
  Leaf
    t.value++
  SubTree
    par
      do_walk (t.left)
      do_walk (t.right)
  Empty
    skip     # optional
```

- Must supply a 'default' variant that is used for initialisation.

Tuples and Abstract Types

- Anonymous structure (tuple) types (allowed generally as L-values):

```
chan({int,char}) c
par
  c ! {42,'x'}
  c ? {x,y}
```

- Abstract types, which provide an equivalent of a union and allow for recursive data structures (without having to abuse the forward-scoping rules):

```
define type Tree
define type Leaf is Tree
  int value
define type SubTree is Tree
  Tree left, right
define type Empty is default Tree

Tree t
case t
  Leaf
    t.value++
  SubTree
    par
      do_walk (t.left)
      do_walk (t.right)
  Empty
    skip    # optional
```

- Must supply a '`default`' variant that is used for initialisation.

Enumerated Types

- A notable omission in occam/occam-pi ...

Enumerated Types

- A notable omission in occam/occam-pi ...

```
define enum Colours
  Red
  Green
  Blue
```

Arrays

- Arrays treated like 'mobile' arrays in occam-pi, so zero elements by default (for unsized array declarations).

```
[]uint128 data
data = [10]uint128

[8]int16 sdata
```

- Array and structure elements accessed *either* with 'dot' or square brackets.
  - constant constructors for both use square brackets:

Arrays

- Arrays treated like 'mobile' arrays in occam-pi, so zero elements by default (for unsized array declarations).

```
[]uint128 data
data = [10]uint128

[8]int16 sdata
```

- Array and structure elements accessed *either* with 'dot' or square brackets.
  - constant constructors for both use square brackets:

```
[]int stuff = [1, 3, 6]
```

Barriers

- Simple barrier types, as we already have in occam-pi:

```
barrier b
par                    # compiler figures out which
  proc_a (b)           # processes are enrolled
  proc_b (b)
  seq
    sync b
```

- Also **phased barriers** for safe (CREW) access to shared state:

Barriers

- Simple barrier types, as we already have in occam-pi:

```
barrier b
par                   # compiler figures out which
  proc_a (b)          # processes are enrolled
  proc_b (b)
  seq
    sync b
```

- Also **phased barriers** for safe (CREW) access to shared state:

```
barrier(2) ph
par
  sync ph(0)
  sync ph(1)    # deadlock
```

Barriers

- Simple barrier types, as we already have in occam-pi:

```
barrier b
par                     # compiler figures out which
  proc_a (b)            # processes are enrolled
  proc_b (b)
  seq
    sync b
```

- Also **phased barriers** for safe (CREW) access to shared state:

```
barrier(2) ph
par
  sync ph(0)
  sync ph(1)     # deadlock
```

```
define foo (barrier(2) x)
  case sync x
    0
      # in phase 0
    1
      # in phase 1
```

Function and Process Types

- For implementing (roughly) an equivalent of C's **function pointer** mechanism.
  - more than just a pointer in practice – memory usage, etc.

Function and Process Types

- For implementing (roughly) an equivalent of C's **function pointer** mechanism.
  - more than just a pointer in practice – memory usage, etc.

```
(val int) -> int fcn
(val int, val int) -> int, int rand_fcn
(barrier, chan!(char)) proc

define type i_to_i is (val int) -> int
```

Processes / Procedures

- Straightforward named blocks of code:

```
define out_10 (val int x, chan!(int) out)
  seq i = x for 10
    out ! i
  # some more code here
```

- Parameter passing uses a **renaming** semantics, so inlining has (logically) no effect.
- For convenience, allow the direction on channels to be specified alongside the name:

Processes / Procedures

- Straightforward named blocks of code:

```
define out_10 (val int x, chan!(int) out)
  seq i = x for 10
    out ! i
  # some more code here
```

- Parameter passing uses a **renaming** semantics, so inlining has (logically) no effect.
- For convenience, allow the direction on channels to be specified alongside the name:

Processes / Procedures

- Straightforward named blocks of code:

```
define out_10 (val int x, chan!(int) out)
  seq i = x for 10
    out ! i
  # some more code here
```

- Parameter passing uses a **renaming** semantics, so inlining has (logically) no effect.
- For convenience, allow the direction on channels to be specified alongside the name:

```
define succ (chan(int) in?, out!)
  while true
    int x
    in ? x
    out ! x+1
```

Functions

- Like occam, functions must be *pure* (no side-effects):

```
define sum (val int data[]) -> int
  int res = 0
  seq i = 0 for size(data)
    res += data[i]
  return res
```

- We'll allow functions to allocate and release memory, on the assumption that the heap is passed-to and returned-from the function.
- Also allow multi-value/multi-typed functions:

Functions

- Like occam, functions must be *pure* (no side-effects):

```
define sum (val int data[]) -> int
  int res = 0
  seq i = 0 for size(data)
    res += data[i]
  return res
```

- We'll allow functions to allocate and release memory, on the assumption that the heap is passed-to and returned-from the function.
- Also allow multi-value/multi-typed functions:

Functions

- Like occam, functions must be *pure* (no side-effects):

```
define sum (val int data[]) -> int
  int res = 0
  seq i = 0 for size(data)
    res += data[i]
  return res
```

- We'll allow functions to allocate and release memory, on the assumption that the heap is passed-to and returned-from the function.
- Also allow multi-value/multi-typed functions:

```
define minmax (val int data[]) -> int, int
  int min = 0, max = 0
  ...  code
  return min, max
```

Expressions

- No operator precedence (like occam), so explicit bracketing:
  - however, to avoid painful bracketing, assume left-to-right evaluation for the same operator

```
int x = (a + b) * (c + 42)
```

- Arithmetic overflow (and underflow) still generate run-time errors.
- Automatic type promotion where required (and obviously harmless), but no automatic coercion or truncation.
- Casting required between different types (e.g. integer and real):

Expressions

- No operator precedence (like occam), so explicit bracketing:
  - however, to avoid painful bracketing, assume left-to-right evaluation for the same operator

```
int x = (a + b) * (c + 42)
a = b + c + x
```

- Arithmetic overflow (and underflow) still generate run-time errors.
- Automatic type promotion where required (and obviously harmless), but no automatic coercion or truncation.
- Casting required between different types (e.g. integer and real):

Expressions

- No operator precedence (like occam), so explicit bracketing:
  - however, to avoid painful bracketing, assume left-to-right evaluation for the same operator

```
int x = (a + b) * (c + 42)
a = b + c + x
```

- Arithmetic overflow (and underflow) still generate run-time errors.
- Automatic type promotion where required (and obviously harmless), but no automatic coercion or truncation.
- Casting required between different types (e.g. integer and real):

Expressions

- No operator precedence (like occam), so explicit bracketing:
  - however, to avoid painful bracketing, assume left-to-right evaluation for the same operator

```
int x = (a + b) * (c + 42)
a = b + c + x
```

- Arithmetic overflow (and underflow) still generate run-time errors.
- Automatic type promotion where required (and obviously harmless), but no automatic coercion or truncation.

```
int16 x, y = 42
uint8 z = 0xff
x = z
z = int16 y
```

- Casting required between different types (e.g. integer and real):

Expressions

- No operator precedence (like occam), so explicit bracketing:
  - however, to avoid painful bracketing, assume left-to-right evaluation for the same operator

```
int x = (a + b) * (c + 42)
a = b + c + x
```

- Arithmetic overflow (and underflow) still generate run-time errors.
- Automatic type promotion where required (and obviously harmless), but no automatic coercion or truncation.

```
int16 x, y = 42
uint8 z = 0xff
x = z
z = int16 y
```

- Casting required between different types (e.g. integer and real):

```
int128 p = some_function (42)
real128 r = real128 trunc p
```

Expressions

- Support for a **conditional** expression, as found in various languages:

```
int v = (y == 42) ? 99 : z
```

- Also support for **lambda** abstractions, assignable to function types:

- These are dealt with at compile-time, compiled into a named function or inlined.

Expressions

- Support for a **conditional** expression, as found in various languages:

```
int v = (y == 42) ? 99 : z
```

- Also support for **lambda** abstractions, assignable to function types:

```
define type i_to_i is (val int) -> int
i_to_i fcn = \v.(v * v)
```

- These are dealt with at compile-time, compiled into a named function or inlined.

Expressions

- Support for a **conditional** expression, as found in various languages:

```
int v = (y == 42) ? 99 : z
```

- Also support for **lambda** abstractions, assignable to function types:

```
define type i_to_i is (val int) -> int
i_to_i fcn = \v.(v * v)
```

- These are dealt with at compile-time, compiled into a named function or inlined.

```
int v = \x.(x * y) 14

define generator (chan!(i_to_i) out)
  out ! \x.(x * (x + 1))
```

Operators

- The usual set of operators as found in occam/occam-pi, with a C flavoured syntax.
- Comparison: ‘<’, ‘<=’, ‘==’, ‘>=’, ‘>’, ‘!=’, ‘<>’
- Boolean logic: ‘&&’, ‘||’, ‘><’, ‘!’
- Bitwise: ‘&’, ‘|’, ‘^’, ‘~’
- Arithmetic (checked): ‘+’, ‘-’, ‘*’, ‘/’, ‘\’, ‘<<’, ‘>>’
- Arithmetic (unchecked): ‘plus’, ‘minus’, ‘times’

Operators

- For convenience support for increment/decrement and similar operators (really *processes*, as they cannot be used as R-values):

```
int x = 42

x++          # x = x + 1
x -= y       # x = x - y
x *= 15      # x = x * 15
```

Flow Control

- Allow '`return`' from any point inside a procedure/function.
  - not a problem for modelling execution as always doable using boolean flags and '`if`'s
  - restricted to *sequential* code (no outstanding parallel processes!)
- Allow '`break`' inside while-loops.
  - undecided: allowing labelled loops, etc. and targetted '`break`'.
- Exception handling: kept straightforward – basic try/catch/finally.
  - again, restricted to purely sequential code.

Flow Control

- Allow '`return`' from any point inside a procedure/function.
  - not a problem for modelling execution as always doable using boolean flags and '`if`'s
  - restricted to *sequential* code (no outstanding parallel processes!)
- Allow '`break`' inside while-loops.
  - undecided: allowing labelled loops, etc. and targetted '`break`'.
- Exception handling: kept straightforward – basic try/catch/finally.
  - again, restricted to purely sequential code.

Flow Control

- Allow '`return`' from any point inside a procedure/function.
  - not a problem for modelling execution as always doable using boolean flags and '`if`'s
  - restricted to *sequential* code (no outstanding parallel processes!)
- Allow '`break`' inside while-loops.
  - undecided: allowing labelled loops, etc. and targetted '`break`'.
- Exception handling: kept straightforward – basic try/catch/finally.
  - again, restricted to purely sequential code.

```
try
  some_routine (x, y, 42)
  some_other_routine (z)
  int8 v = int8 trunc 3.14159
catch
  report_error ()
finally
  cleanup ()
```

Primitive Processes

- Usual two suspects, '`skip`' and '`stop`':
  - use of '`skip`' is largely optional — indentation rules mean it's obvious when it's missing.
  - '`stop`' is the traditional self deadlock.
- Also '`abort`', which is captured within a '`try`' block, else run-time error.
- Built-in '`assert`' primitive produces a run-time error if triggered, uncatchable.

Primitive Processes

- Usual two suspects, '`skip`' and '`stop`':
  - use of '`skip`' is largely optional — indentation rules mean it's obvious when it's missing.
  - '`stop`' is the traditional self deadlock.
- Also '`abort`', which is captured within a '`try`' block, else run-time error.
- Built-in '`assert`' primitive produces a run-time error if triggered, uncatchable.

Primitive Processes

- Usual two suspects, '`skip`' and '`stop`':
  - use of '`skip`' is largely optional — indentation rules mean it's obvious when it's missing.
  - '`stop`' is the traditional self deadlock.
- Also '`abort`', which is captured within a '`try`' block, else run-time error.
- Built-in '`assert`' primitive produces a run-time error if triggered, uncatchable.

Structured Processes

- 'if', 'alt', 'seq' and 'par' almost as they are in occam.
- 'case' and 'while' too.

```
if
  x == 42
    do_something ()
  x < y
    do_something_else ()
  true
    assert x >= y
```

- Can nest and replicate the first four in the same way as occam.
  - 'seq' and 'par' can omit replicator name if not needed:

Structured Processes

- 'if', 'alt', 'seq' and 'par' almost as they are in occam.
- 'case' and 'while' too.

```
if
  x == 42
    do_something ()
  x < y
    do_something_else ()
  true
    assert x >= y
```

```
alt
  in[0] ? x
    out ! x
  in[1] ? x
    out ! x
  (n > 16) & c ? x
    out ! x
```

- Can nest and replicate the first four in the same way as occam.
  - 'seq' and 'par' can omit replicator name if not needed:

Structured Processes

- 'if', 'alt', 'seq' and 'par' almost as they are in occam.
- 'case' and 'while' too.

```
if
  x == 42
    do_something ()
  x < y
    do_something_else ()
  true
    assert x >= y
```

```
alt
  in[0] ? x
    out ! x
  in[1] ? x
    out ! x
  (n > 16) & c ? x
    out ! x
```

```
seq
  do_this ()
  then_that ()
```

- Can nest and replicate the first four in the same way as occam.
  - 'seq' and 'par' can omit replicator name if not needed:

Structured Processes

- 'if', 'alt', 'seq' and 'par' almost as they are in occam.
- 'case' and 'while' too.

```
if
  x == 42
    do_something ()
  x < y
    do_something_else ()
  true
    assert x >= y
```

```
alt
  in[0] ? x
    out ! x
  in[1] ? x
    out ! x
  (n > 16) & c ? x
    out ! x
```

```
seq
  do_this ()
  then_that ()
```

```
par
  receiver (c?)
  sender (c!)
```

- Can nest and replicate the first four in the same way as occam.
  - 'seq' and 'par' can omit replicator name if not needed:

Structured Processes

- 'if', 'alt', 'seq' and 'par' almost as they are in occam.
- 'case' and 'while' too.

```
if
  x == 42
    do_something ()
  x < y
    do_something_else ()
  true
    assert x >= y
```

```
alt
  in[0] ? x
    out ! x
  in[1] ? x
    out ! x
  (n > 16) & c ? x
    out ! x
```

```
seq
  do_this ()
  then_that ()
```

```
par
  receiver (c?)
  sender (c!)
```

- Can nest and replicate the first four in the same way as occam.
  - 'seq' and 'par' can omit replicator name if not needed:

```
seq for 10
  do_something ()
```

Structured Processes

- Allow a shorter version of '`if`' for more convenient uses:

```
if x == 42
  do_something ()
```

- Also inline versions of '`seq`' and '`par`':

- Inline '`seq`' ('`->`' read *then*) can also be used in '`alt`' constructs for brevity:

Structured Processes

- Allow a shorter version of ‘`if`’ for more convenient uses:

```
if x == 42
  do_something ()
```

- Also inline versions of ‘`seq`’ and ‘`par`’:

```
c ! 42 ||| c ? y
screen ! 'c' -> screen ! '\n'
```

- Inline ‘`seq`’ (‘`->`’ read *then*) can also be used in ‘`alt`’ constructs for brevity:

Structured Processes

- Allow a shorter version of 'if' for more convenient uses:

```
if x == 42
  do_something ()
```

- Also inline versions of 'seq' and 'par':

```
c ! 42 ||| c ? y
screen ! 'c' -> screen ! '\n'
```

- Inline 'seq' ('->' read *then*) can also be used in 'alt' constructs for brevity:

```
pri alt
  c ? x -> out ! x+1
  d ? y -> stop -> skip
```

Channel Mobility

- An important feature for many applications
  - least not complex systems simulations!
- Ordinary channels cannot have their ends pulled apart; mobile channels must be constructed explicitly:
- Higher-order channels are straightforward (and consistent) :-)

Channel Mobility

- An important feature for many applications
  - least not complex systems simulations!
- Ordinary channels cannot have their ends pulled apart; mobile channels must be constructed explicitly:

```
chan?(int) c
chan!(int) d
bind c?, d!

bind chan(char) e?, f!
```

- Higher-order channels are straightforward (and consistent) :-)

Channel Mobility

- An important feature for many applications
  - least not complex systems simulations!
- Ordinary channels cannot have their ends pulled apart; mobile channels must be constructed explicitly:

```
chan?(int) c
chan!(int) d
bind c?, d!

bind chan(char) e?, f!
```

- Higher-order channels are straightforward (and consistent) :-)

```
chan?(int) c
chan!(chan?(int)) d
chan?(chan!(chan?(int))) e
e ? d
d ! c
```

User Defined Operators

- Essentially **operator overloading**, generally a useful language feature (added to occam by Jim Moores).
  - allowed as part of type definitions for that type (as well as stand-alone).
  - must follow rules for functions (no side-effects!).

```
define type ICoord
  int x, y
```

User Defined Operators

- Essentially **operator overloading**, generally a useful language feature (added to occam by Jim Moores).
  - allowed as part of type definitions for that type (as well as stand-alone).
  - must follow rules for functions (no side-effects!).

```
define type ICoord
  int x, y

  "+" (val a, b) -> ICoord
    ICoord r
    r.x = a.x + b.x
    r.y = a.y + b.y
    return r
```

User Defined Operators

- Essentially **operator overloading**, generally a useful language feature (added to occam by Jim Moores).
  - allowed as part of type definitions for that type (as well as stand-alone).
  - must follow rules for functions (no side-effects!).

```
define type ICoord
  int x, y

  "+" (val a, b) -> ICoord
    ICoord r
    r.x = a.x + b.x
    r.y = a.y + b.y
    return r

  "-" (val a, b) -> ICoord = [a.x - b.x, a.y - b.y]
```

User Defined Operators

- Essentially **operator overloading**, generally a useful language feature (added to occam by Jim Moores).
  - allowed as part of type definitions for that type (as well as stand-alone).
  - must follow rules for functions (no side-effects!).

```
define type ICoord
  int x, y

  "+" (val a, b) -> ICoord
    ICoord r
    r.x = a.x + b.x
    r.y = a.y + b.y
    return r

  "-" (val a, b) -> ICoord = [a.x - b.x, a.y - b.y]

define sq (val int v) -> int = (v * v)
define "<->" (val ICoord x, y) -> int
  return sq (x.x-y.x) + sq (x.y-y.y)
```

Type Inference and Polymorphism

- Allow the compiler to figure out the return type of a function (less typing for the programmer):

```
define foo (val int a, b)
  int pl, mi
  pl = a + b ||| mi = a - b
  return pl, mi
```

- Functions and procedures may have generic types.
  - specialised by the compiler for specific types:

Type Inference and Polymorphism

- Allow the compiler to figure out the return type of a function (less typing for the programmer):

```
define foo (val int a, b)
  int pl, mi
  pl = a + b ||| mi = a - b
  return pl, mi

define foo (val int a, b) = a + b, a - b
```

- Functions and procedures may have generic types.
  - specialised by the compiler for specific types:

Type Inference and Polymorphism

- Allow the compiler to figure out the return type of a function (less typing for the programmer):

```
define foo (val int a, b)
  int pl, mi
  pl = a + b ||| mi = a - b
  return pl, mi
define foo (val int a, b) = a + b, a - b
```

- Functions and procedures may have generic types.
  - specialised by the compiler for specific types:

```
define id (chan(T) in?, out!)
  while true
    T v
    in ? v -> out ! v
```

Var-Args and Run-Time Type Selection

- **Disclaimer**: this is not necessarily concrete yet!
- Support an explicit '`type`' type, useful for run-time decision making:

```
define typeset vararg is int, byte, uint, string

define printf (chan!(char) out, string fmt, []vararg args)
  ...  stuff
  seq i = 0 for size args
    case typeof args[i]
      int
        ...  code for integer
      string
        ...  code for string
      else
        ...  unhandled cases
```

Dining Philosophers

Dining Philosophers

```
define main (chan!(char) screen)
  par
    # display stuff here...
    secure_college ()
```

Dining Philosophers

```
define main (chan!(char) screen)
  par
    # display stuff here...
    secure_college ()
```

```
define secure_college ()
  [5]chan() left, right
  [5]chan() up, down
  par
    par i = 0 for 5
      philosopher (up[i]!, down[i]!, left[i]!, right[i]!)
    par i = 0 for 5
      fork (left[i]?, right[(i+1)\5]?)
    security (down?, up?)
```

Dining Philosophers

```
define fork (chan() left?, right?)
  while true
    alt
      left? -> left?
      right? -> right?
```

Dining Philosophers

```
define fork (chan() left?, right?)
  while true
    alt
      left? -> left?
      right? -> right?
```

```
define philosopher (chan() up!, down!, fork_left!, fork_right!)
  while true
    # think ...
    down!
    fork_left! ||| fork_right!
    # eat ...
    fork_left! ||| fork_right!
    up!
```

Dining Philosophers

```
define security ([]chan() downs?, ups?)
  int sat = 0
  val int limit = 4
  while true
    alt
      alt i = 0 for size(downs)
        (sat < limit) & downs[i]?
          sat++
      alt i = 0 for size(ups)
        ups[i]?
          sat--
```

Other Things

- For two-way protocols specifically, '`chan+`' and '`chan-`' for client and server sides.
- Compiler extensions to allow experimentation with language structure and similar.
- A sensible **module** system for building libraries.
- Bindings to interface with existing C and occam-pi code.
- Low-level things such as '`placed`' data and '`port`'s.
- And probably a whole lot of other things...!

State of Things

- Mostly ideas at the moment, but slowly forming into something concrete and reasonable
  - suggestions for things to add, remove or modify very welcome!
  - goal is to produce a safe concurrent language that is quick and easy to use (without compromising existing run-time performance)
- Some bits of a compiler in place in the NOCC compiler framework
  - generating mostly empty LLVM files at the moment, but in progress!