Mobile Data Types for Communicating Processes

Peter Welch
Computing Laboratory
University of Kent at Canterbury
(P.H.Welch@ukc.ac.uk)

PDPTA 2001, Las Vegas, Nevada (28th. June, 2001)

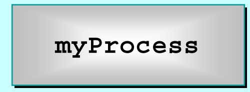
Communicating Sequential Processes (CSP)

CSP deals with ***processes***, ***networks*** of processes and various forms of ***synchronisation*** / ***communication*** between processes.

A network of processes is also a process - so CSP naturally accommodates layered network structures (***networks of networks***).

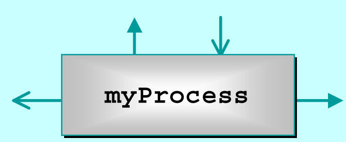
We do not need to be mathematically sophisticated to work with CSP. That sophistication is pre-engineered into the model. We benefit from this simply by using it.

Processes



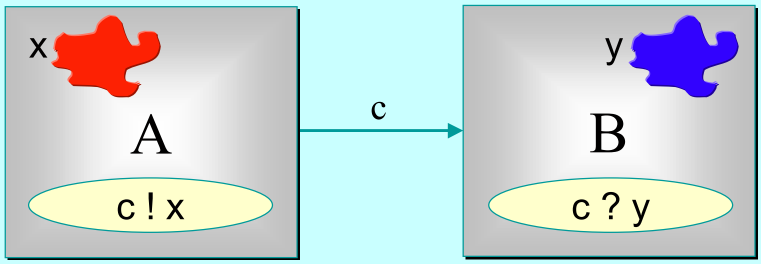
- A **process** is a component that encapsulates some data structures and algorithms for manipulating that data.
- Both its data and algorithms are **private**. The outside world can neither see that data nor execute those algorithms! [It is not an *object*.]
- The algorithms are executed by the process in its own thread (or threads) of control.
- So, how does one process interact with another?

Processes



- The simplest form of interaction is *synchronised* message-passing along **channels**.
- The simplest forms of channel are **zero-buffered** and **point-to-point** (i.e. *wires*).
- But, we can have **buffered** channels (*blocking*/*overwriting*).
- And **any-1**, **1-any** and **any-any** channels.
- And structured multi-way synchronisation (e.g. **barriers**) …
- And high-level (e.g. **CREW**) *shared-memory* locks …

Synchronised Communication

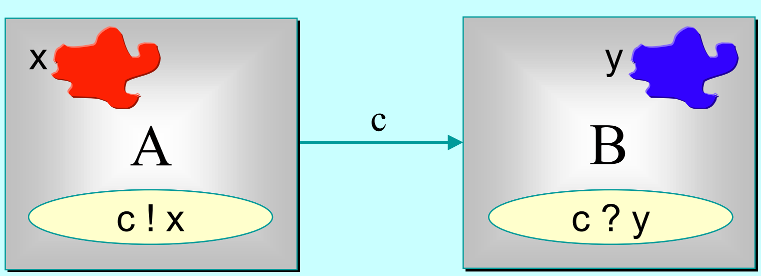


A may write on ***c*** at any time, but has to wait for a *read*.

B may read from ***c*** at any time, but has to wait for a *write*.

(A (c) || B (c)) \ {c}

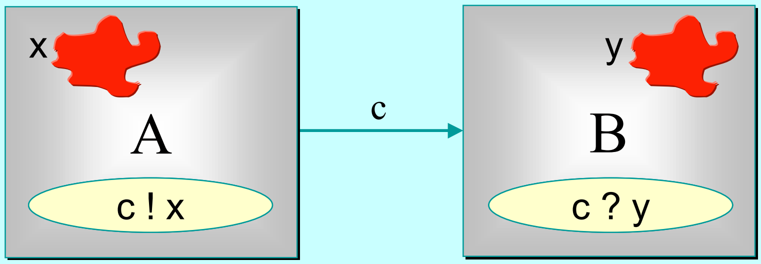
Synchronised Communication



Only when both ***A*** and ***B*** are ready can the communication proceed over the channel ***c …***

(A (c) || B (c)) \ {c}

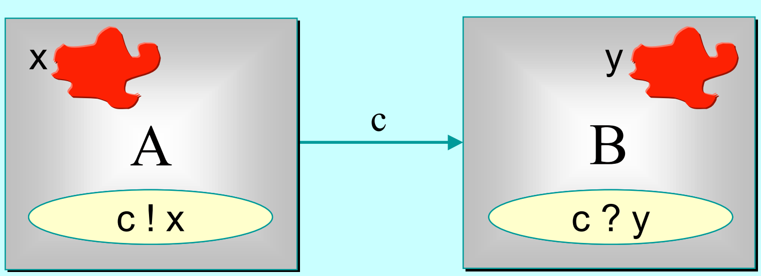
Synchronised Communication



Only when both ***A*** and ***B*** are ready can the communication proceed over the channel ***c … and it's happened!***

(A (c) || B (c)) \ {c}

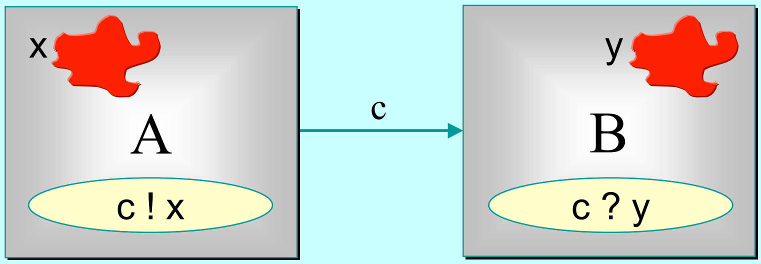
Copy Semantics



Classical occam: data are *copied* from the workspace of ***A*** into the workspace of ***B***. Subsequent work by ***A*** on its ***x*** variable and ***B*** on its ***y*** variable causes no mutual interference.

SAFE but SLOW

Reference Semantics

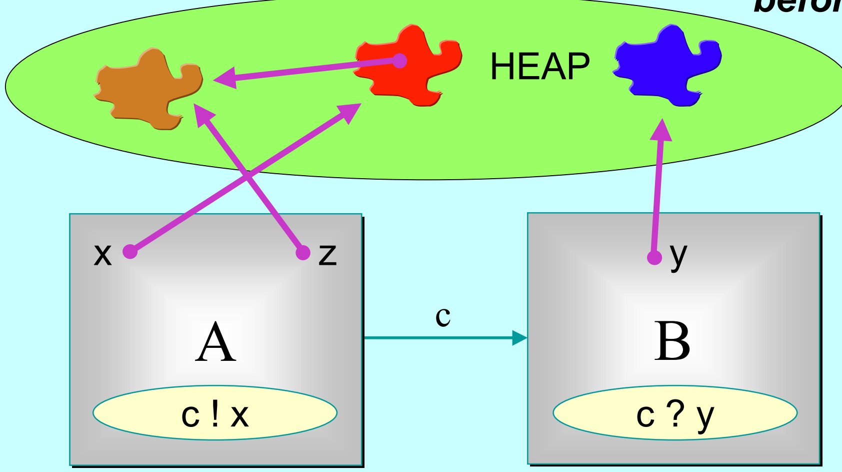


Java/JCSP: *references* to data (objects) are copied from the workspace of ***A*** into ***B***. Subsequent work by ***A*** on its ***x*** variable and ***B*** on its ***y*** variable causes mutual interference (*race hazard*).

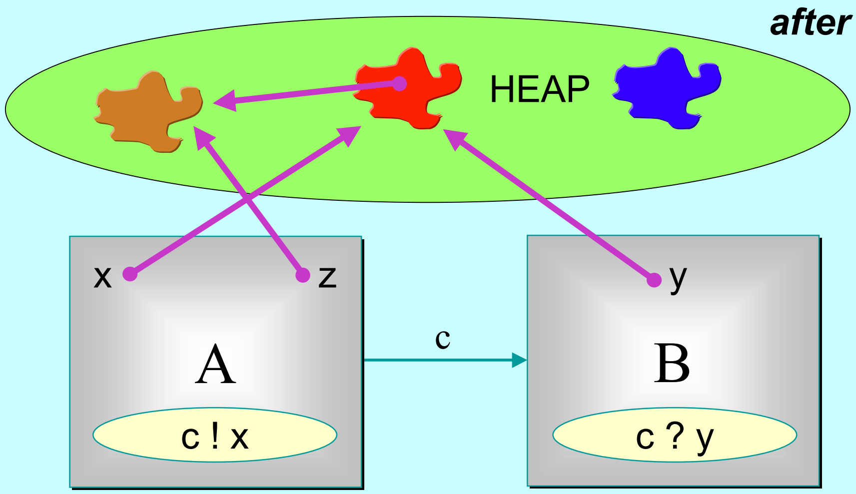
UNSAFE but FAST

Reference Semantics

before

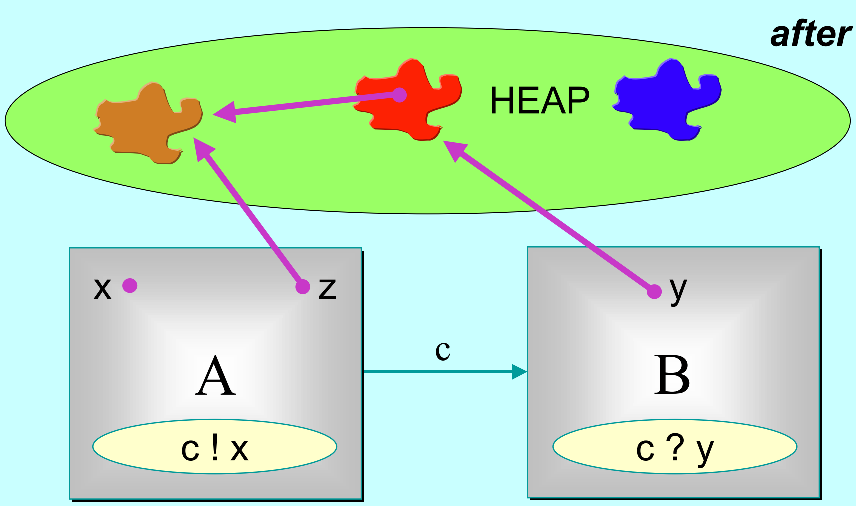


Reference Semantics



Red and brown objects are parallel compromised!

Reference Semantics



Even if the source variable is *nulled*, brown is done for!!

Classical occam

Different in-scope variables ***implies*** different pieces of data (*zero aliasing*).

Automatic guarantees against ***parallel race hazards*** on data access … and against ***serial aliasing accidents***.

Overheads for ***large*** data communications:

- space (needed at both ends for both copies);
- time (for copying).

Java / JCSP

Hey … it's Java … so ***aliasing*** is endemic.

No guarantees against ***parallel race hazards*** on data access … or against ***serial aliasing accidents***. We must look after ourselves.

Overheads for ***large*** data communications:

- space (***shared*** by both ends);
- time is O(1).

What to do?

Build tools to make **Java** secure against aliasing errors (and maintain fast communications through reference sharing)?

Too hard … probably impossible …

Make **occam** more flexible. Let it communicate references (and keep it secure against aliasing errors)?

This is what we have done …

The Trick Is …

Between processes is ***distinct*** memory spaces, communication ***has to be*** done by copying the data.

Between processes in the ***same*** memory space, communication ***may be*** done by copying references to the data.

The trick is to make the above two scenarios semantically compatible … and the latter one semantically safe.

Mobile Semantics

Consider ***copy*** and ***move*** operations on files …

copy duplicates the file, placing the copy in the target directory under a (possibly) new name.

move moves the file to the target directory, possibly renaming it:

- the original file can no longer be found at the source address;
- it has *moved*.

Copy Assignment

As assignment changes the state of its environment, which we can represent by a set of ordered pairs mapping variables into data values.

In **occam**, because of its zero-tolerance of aliasing, assignment semantics is what we expect:

$\{ \langle x_0, v_0 \rangle, \langle x_1, v_1 \rangle, \ldots \}$ `x0 := x1` $\{ \langle x_0, v_1 \rangle, \langle x_1, v_1 \rangle, \ldots \}$

[In all other languages with assignment, the semantics are much more complex - since the variable `x0` may be aliased to other variables … and the values associated with those aliases will also have changed to v_1.]

Mobile Assignment

Here again is the standard ***copy*** semantics for assignment:

$\{ \langle x_0, v_0 \rangle, \langle x_1, v_1 \rangle, \ldots \}$ `x0 := x1` $\{ \langle x_0, v_1 \rangle, \langle x_1, v_1 \rangle, \ldots \}$

Mobile semantics differs in one crucial respect:

$\{ \langle x_0, v_0 \rangle, \langle x_1, v_1 \rangle, \ldots \}$ `x0 := x1` $\{ \langle x_0, v_1 \rangle, \langle x_1, ?? \rangle, \ldots \}$

The value of the variable at the source of the assignment has become *undefined* - its value has ***moved*** to the target variable.

Note: mobile semantics is ***strictly weaker*** than copy semantics. Indeed the latter is a perfectly legal implementation of the former - a fact of which we shall take advantage presently.

Mobile Communication

The semantics for assignment and communication are directly related. In **occam**, communication is just a distributed form of assignment - a value computed in the sending process is assigned to a variable in the receiving one (after the two processes have synchronised).

For example, if `x0` and `x1` were of type `FOO`, we have the semantic equivalence:

```
x0 := x1
```

equals

```
CHAN OF FOO c:
PAR
  c ! x1
  c ? x0
```

We preserve this equivalence for mobile communications - hence, *the value of the sent mobile variable becomes undefined.*

Mobile Syntax (1/4)

We propose two extra keywords for **occam** - a `MOBILE` qualifier for data types/variables and a `CLONE` prefix operator.

The `MOBILE` qualifier doesn't change the semantics of types *as types*. For example, `MOBILE` types are compatible with ordinary types in expressions.

But it imposes the ***mobile semantics*** on assignment and communication between `MOBILE` variables. So, if we had:

```
DATA TYPE FOO
  MOBILE RECORD
    ... <data fields>
:

FOO x0, x1:
```

… then assignment and communication between `x0` and `x1` has ***mobile*** (not ***copy***) ***semantics***.

Mobile Syntax (2/4)

The `MOBILE` qualifier need not be burnt into the type declaration - it can be associated just with particular variables.

For example, the following is an alternative to the declarations on the previous slide:

```
DATA TYPE BAR
  RECORD
    ... <same fields as FOO>
:

MOBILE BAR x0, x1:
```

… then assignment and communication between `x0` and `x1` has ***mobile*** (not ***copy***) ***semantics***.

Mobile Syntax (3/4)

In some cases, we may need ***copy*** semantics from `MOBILE` variables. For this purpose, a `CLONE` operator is provided. This generates a ***copy*** of the mobile on which the required assignment or communication can be performed.

For example:

```
SEQ
  x0 := CLONE x1
  c ! CLONE x1
```

`x0` and `x1` are ***mobile*** variables

Both the above assignment and communication leave the value of `x1` alone - i.e. we are back to ***copy*** semantics.

Mobile Syntax (4/4)

We had earlier toyed with having new symbols to indicate ***mobile assignment*** (`<-`) and ***mobile output*** (`<!`).

Then, we could use the ordinary symbols (`:=` and `!`) to mean ***copy assignment*** and ***output*** and there would be no need for the `CLONE` operator.

But that would prevent us having mixed mobile and non-mobile components in messages. For example, if:

```
PROTOCOL MIXED IS FOO; BAR; FOO:
CHAN OF MIXED mixed:
BAR y:
```

Then:

```
mixed ! x0; y; x0
```

… leaves `x0` and `x1` *undefined* - but `y` is *unchanged*

Undefined Usage Checks (1/3)

It is an ***error*** to look at a variable whose data value is currently *undefined*.

This is exactly the same ***error*** as attempting to access the value of an *uninitialised* variable (mobile or non-mobile).

We have modified the KRoC **occam** compiler to analyse the state (*defined* or *undefined*) of all variables at all points of use. We track across function/procedure boundaries, including those from separately compiled libraries. Remember that **`MOBILE`** variables, unlike ordinary ones, can transition both ways between *undefined* and *defined*.

We take a conservative view: if there is a *runtime-decided* path through some serial logic (**`SEQ`**/**`IF`**/**`CASE`**/**`WHILE`**/**`ALT`**) that can leave a variable *undefined*, it is marked *undefined.*

Undefined Usage Checks (2/3)

Undefined usage checks take place following parallel usage checks. So, if a variable is *undefined* before a **`PAR`** construct, it remains *undefined* if no single parallel component defines it.

A more difficult problem is tracking the status of array elements, whose indices are usually run-time values. There is a similar problem for the existing parallel usage checker. We adopt the same solution: treat the array as an ***atomic unit*** - either wholly *defined* or *undefined*. This may get more sophisticated later.

Tracking the status of record fields is not a problem. Each <*variable, field-name*> pair is treated as a separate variable.

Undefined Usage Checks (3/3)

Currently, our compiler only issues warnings about the use of *undefined* (or *possibly undefined*) variables. Later, this will change to rejection.

For well-designed code, the conservative nature of the analysis should cause no problem. For bad code, compiler rejection will encourage better style.

[An alternative to *definedness checking* would be to define ***default values*** for all types and set them following variable declaration or mobile operations. If the default values were ***type-illegal***, their use could be trapped at run-time. We are not doing this. ***Legal*** default values (such as zero) causes errors for lazy programmers. ***Illegal*** ones lead to needless run-time overheads.]

Implementation of Mobiles

As said earlier, implementation of mobile operations by copying is perfectly legal. For efficiency, this is precisely how *small* mobiles (e.g. those whose data types require less than 8 bytes) are managed - *the compiler simply ignores the* **`MOBILE`** *qualification*.

The same can be done for mobile communications between processes occupying *different* memory spaces.

The interesting case is mobile communications between processes occupying *the same* memory space. And, of course, for mobile assignments (which can only be within *the same* memory space).

Mobiles in the Same Memory

The obvious scheme is used: **`MOBILE`** variables hold pointers to their actual data. Those pointers are not apparent to the user (the same as for **Java**). Mobile assignment/communication is just the copying of those pointers.

However, unlike OO languages, we are not going to allow these pointers to set up any ***aliases***.

Space for all mobile data will be in a globally accessible heap (***mobilespace***). Unlike conventional heaps, strict ***zero-aliasing*** will be conserved. It will hold only tree structures and **`MOBILE`** variables will only point to the root of such trees. Different **`MOBILE`**s will reference different trees.

There are No Null Mobiles!

Although **`MOBILE`** variables hold pointers, we have decided against allowing the user to know about or see **`NULL`** values (unlike **Java**). Correct code would never try to follow **`NULL`** pointers - the undefined analysis will see to that.

In fact, we will arrange that mobile variables hold at all times ***valid*** pointers - although the data at which they are pointed may sometimes be ***undefined***. The undefined analysis will prevent following those pointers in the latter case.

Pre-Allocated Mobilespace

Classical **occam** has constraints designed to meet security requirements for embedded systems operating within fixed size (sometimes very small) memory limits:

Forbidden are recursion, runtime computed parallel replication counts and runtime sized arrays.

Sticking to these constraints allows tremendous optimisation in the management of ***mobilespace***.

The total number of **`MOBILE`** variables (record fields, array elements) that can ever become active in a system is known to the compiler - plus the sizes of all types underlying those mobiles. Therefore, the total size for ***mobilespace*** can be exactly calculated and pre-initialised.

(Free-List Mobilespace)

The original plan was to maintain free-lists of mobile data nodes - one for each underlying type. The maximum size of those free-lists would be known in advance.

When a **`MOBILE`** loses data because it *receives* new data by mobile assignment/communication - or because it *exits scope* - the lost data node is appended to the relevant free-list.

When a **`MOBILE`** loses data because it is the *source* of a mobile assignment/communication, pick up some undefined material from the relevant free-list.

Both these operations are unit time.

Swapping Mobilespace

The free-lists are not needed!

When a MOBILE loses its data because it receives new data from a mobile assignment or communication, park the lost data node with the mobile variable at the source of the operation.

That sorts out the MOBILE source and target at the same time - *they simply swap pointers!* The fixed size of ***mobilespace*** is conserved, along with ***zero-aliasing***.

> Formally, this implements the mobile semantics. Data has moved from *source* to *target* and the *source* variable has become *undefined*.

The fact that the *source* now holds the *target*'s old data (i.e. that the transfer is bi-directional), we ***forget***. No advantage must be taken of this - to allow the *copying* implementations for small mobiles and between memory spaces.

Mobile Storage Allocation (1/7)

The pointers to mobile data cannot safely reside in ordinary process ***workspace***. That gets reused by other processes and we must not lose the referenced nodes.

Instead, pointers to mobile nodes must reside (permanently) in ***mobilespace***, along with the nodes themselves.

Each MOBILE variable must have a shadow in ***mobilespace*** holding a pointer to a mobile node. When the MOBILE comes into scope, it picks up the pointer held by its shadow. When it leaves scope, it returns its pointer to its shadow.

The returned pointer will (probably) be different if mobile operations have been performed.

Mobile Storage Allocation (2/7)

The compiler generates a static mapping of all mobile data nodes and shadow pointers on to ***mobilespace***. This is similar to how process ***workspace*** and ***vectorspace*** is allocated. The rules for ***mobilespace*** differ in that space cannot be shared between **`SEQ`**uential processes.

Processes using **`MOBILE`** variables are passed an extra parameter, giving the offset into ***mobilespace*** to find its shadows. This is the same mechanism used for ***workspace***.

The compiler generates code to clear all shadow pointers to ***null*** (zero). It also generates code so that the first time a process with **`MOBILE`**s executes, it checks to see if the first shadow pointer is ***null***. If so, it sets up correct pointers to the mobile data blocks already allocated.

Mobile Storage Allocation (3/7)

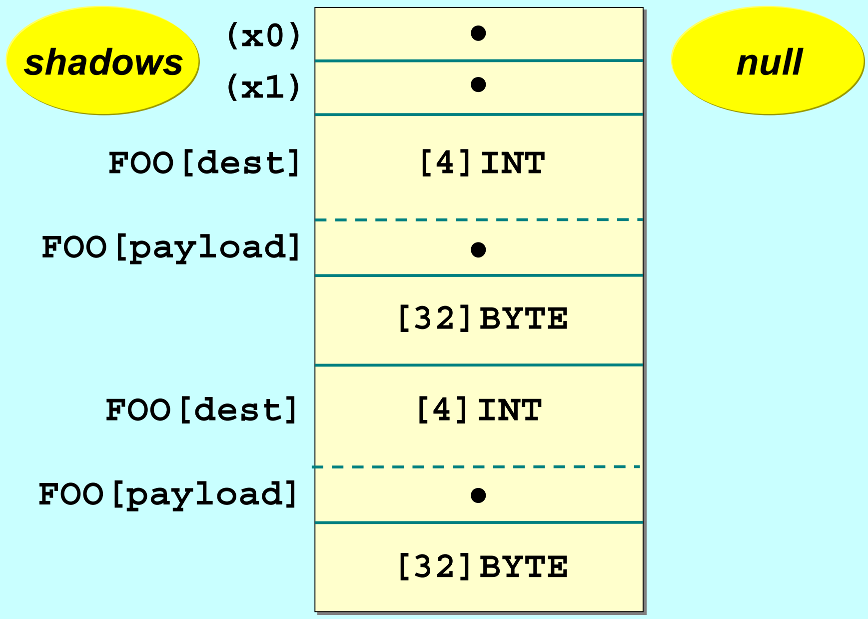
For example, expanding an earlier definition:

```
DATA TYPE FOO
  MOBILE RECORD
    [4]INT dest:
    MOBILE [32]BYTE payload:
:
```

Consider a process containing the following declaration:

```
FOO x0, x1:
```

The first time it comes into scope, it will find its part of ***mobilespace*** in the following state:



Mobile Storage Allocation (5/7)

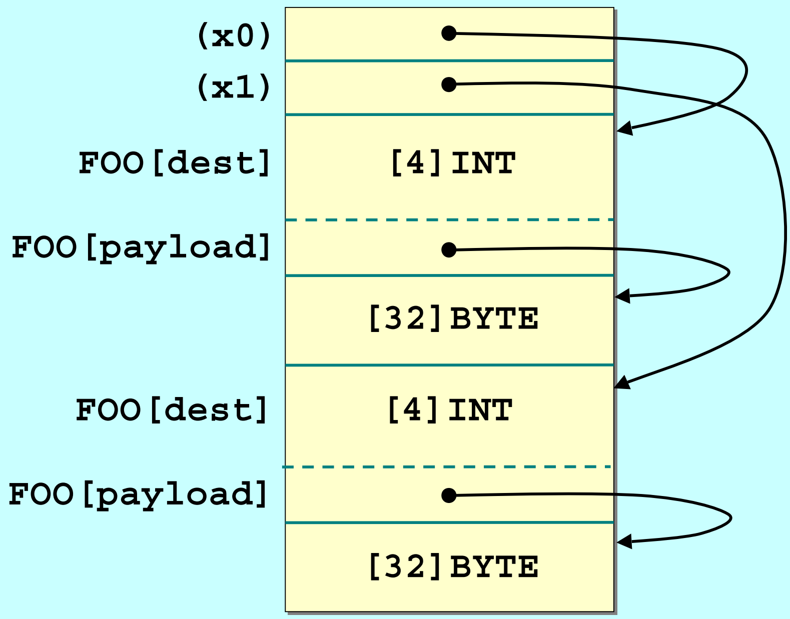
For example, expanding an earlier definition:

```
DATA TYPE FOO
  MOBILE RECORD
    [4]INT dest:
    MOBILE [32]BYTE payload:
:
```

Consider a process containing the following declaration:

```
FOO x0, x1:
```

So, when it comes into scope, if its first **MOBILE** is ***null***, it sets up the pointers correctly to the pre-allocated nodes:



Mobile Storage Allocation (7/7)

For example, expanding an earlier definition:

```
DATA TYPE FOO
  MOBILE RECORD
    [4]INT dest:
    MOBILE [32]BYTE payload:
:
```

Consider a process containing the following declaration:

```
FOO x0, x1:
```

The second and subsequent times it comes into scope, it will find its shadows with valid pointers and simply copies them into **x0** and **x1**. When it leaves scope, it copies them back.

Dynamically Sized Mobiles (1/2)

So far, in keeping with classical **occam**, all mobiles have had statically determined memory requirements.

On systems with no memory constraints, such as those with virtual memory support, one other mobile structure becomes possible - the ***runtime sized*** array:

```
MOBILE []BYTE buffer:
INT n:
SEQ
  in ? n
  buffer := [n]BYTE
  ... process using buffer
```


Dynamically Sized Mobiles (2/2)

Mobile operations on ***dynamic mobiles*** will not be implemented by the *pointer swapping* technique described earlier - since *source* and *target* arrays may not have the same size.

Instead, free-lists of nodes (with separate lists for *half-power-of-two* sizes) are used, combined with Brinch Hansen's algorithm for workspace allocation to support efficient parallel recursion. These form ***dynamic mobilespace***.

Mobile dynamic array nodes are extracted from free-lists on scope entry and returned on scope exit – no *shadows* are therefore needed. The free-lists are also used when nodes are lost and re-acquired during mobile operations. All these are unit time operations.

Mobile Parameters (1/2)

Parameter passing is just ***renaming*** - at least that's the formal position in **occam**. It is different from assignment and communication. So, no mobile ***semantic*** issues arise.

For instance, using **`MOBILE`**s in expressions (as *function* or *user-defined operator* arguments) does not lose their values. Recall that **occam** *functions* and *operators* are guaranteed free from side-effect, *so no mobile assignments can be performed*. Communications cause external state change and are always banned – *mobile or otherwise*.

`MOBILE` variables passed by *reference* to procedures (**occam** **`PROC`**s) may, of course, be moved to another **`MOBILE`** variable or down a channel - no problem.

Mobile Parameters (2/2)

We do not allow functions or operators to declare formal **`VAL MOBILE`** parameters. However, **`MOBILE`** arguments may be passed to formal **`VAL`** parameters of the same underlying type.

For the same reason, in **`PROC`**s we do not allow formal **`VAL MOBILE`** parameters. However, reference **`MOBILE`** parameters are allowed. The following table shows the allowed matches:

	(formal parameter)		
(actual parameter)	**`THING`**	**`VAL THING`**	**`MOBILE THING`**
`THING`	***yes***	***yes***	***no***
`VAL THING`	***no***	***yes***	***no***
`MOBILE THING`	***yes***	***yes***	***yes***

Mobile Performance

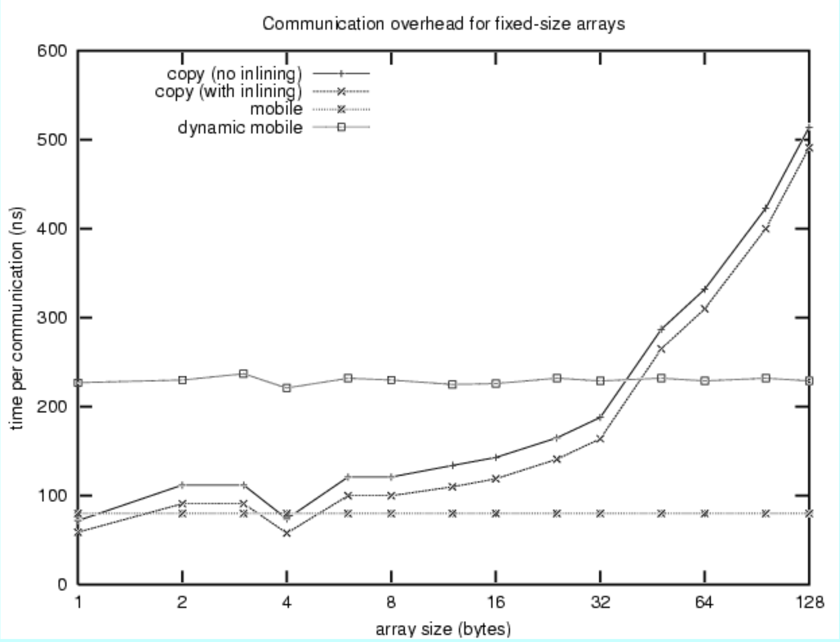
The following figure shows communication times for a simple *producer-consumer* network (running on an 800 MHz. P3).

Two curves show the times (*optimised/non-opt*) for fixed-size ***ordinary*** arrays - the array sizes range from 1 to 128 bytes.

Another curve shows the times for ***fixed-sized mobile*** arrays - again with array sizes range from 1 to 128 bytes.

The final curve shows the times for ***dynamic-sized mobile*** arrays - again with array sizes range from 1 to 128 bytes.

The times include all the overheads for communication - including the two context switches from *producer* to *consumer* and back again. All timings are in ***nanoseconds***!



Status of KRoC Mobiles

Fixed-sized *non-nested* **`MOBILE`** types and variables - ***done***.

This includes the described ***mobilespace*** storage allocation, mobile assignment, mobile communication and parameter passing.

Dynamic-sized *non-nested* **`MOBILE`** arrays - ***done***.

This includes Brinch-Hansen allocation using free-lists for the ***dynamic mobilespace***, plus the usual mobile assignment, mobile communication and parameter passing.

Undefined usage checks - ***done***.

Nested **`MOBILE`** types and variables - ***not yet done***.

Conclusions (1/4)

We have introduced ***mobile communications*** that *move* data from a *source* process (which, therefore, loses it) to a *target* process.

Implementation is ***fast*** (mainly just pointer swapping), ***secure*** (no aliasing is introduced) and ***consistent*** with communication between *different* memory spaces.

To nail the aliasing problem, ***mobile assignment*** has also been introduced - with complementary *movement* semantics and ***fast*** and ***secure*** implementation (again just pointer swapping - apart from dynamic mobiles).

The trick sought at the beginning has been achieved!

Conclusions (2/4)

Repeating this trick for OO languages (such as **Java, C#** or **C++**) is not possible. We could get most of the semantics and fast implementation, but we cannot enforce control of aliasing and make it secure. This is the position for **Java/JCSP**, where we rely on the user knowing the dangers.

OO language change has to happen - too many concepts are missing resulting in serious insecurities. One thing is to separate, by good language engineering, the different uses to which pointers are put.

They should remain hidden, but we should distinguish their use for *sharing information* between different parts of a system (as for **`MOBILE`**s) and for *building interesting data-structures* (such as doubly-linked lists).

Conclusions (3/4)

No time here to describe applications - but they are prolific.

Whenever we have the pattern of accessing some data, processing it and passing it on, these ideas of **`MOBILE`**s are relevant. That pattern is pretty normal.

We have applied this to **`occweb`** - an **occam** *web server* offering highly concurrent performance (and built in one day!).

We are working on a full *graphics/GUI* library for **occam**, where all GUI events and drawing commands map to channel communications. There is very high traffic and almost all commicated packets can be declared **`MOBILE`**. Currently, we are secure - but we do an awful lot of copying!

Conclusions (4/4)

The **MOBILE** pattern is endemic throughout OO systems and most industrial scale applications of **occam** (sadly from past years).

But there is no automated secure management of that pattern and we must take great care - especially when multithreading comes into the equation. Very often, we fail with that care.

This paper contributes to the automation of that care and a considerable reduction in the cost of its execution.