Alan Chalmers, Majid Mirmehdi and Henk Muller (Eds.)
IOS Press, 2001

Copying, Moving and Borrowing semantics

David May Henk Muller
Department of Computer Science, University of Bristol, UK
http://www.cs.bris.ac.uk/

Abstract. In this paper we discuss primitives for mobilising code and communications. We distinguish three types of semantics for mobility: copying (where an identical copy is created remotely), moving (where the original is destroyed), and borrowing (where the original is moved to the target and back to where it came from at defined moments). We discuss these semantics for mobile code and mobile channels. We have implemented Icarus, a language that uses borrowing semantics for mobile code (the *on-statement*) and moving semantics for mobile channels (*first class channels*).

1 Introduction

We have developed the mobile programming language Icarus. Even though we have adopted the C-syntax, Icarus is essentially Occam-based and is used as a vehicle to experiment with migration of code and channels.

Occam has a static process graph, statically distributed over a number of processors. Processes communicate via channels, the communication graph is statically determined and allows highly efficient communication between fine grained processes. In Icarus we have introduced flexibility: the location of computations, and the destinations of communications can be changed dynamically.

In the rest of this paper we use the following notations:

- Scopes are declared using curly braces `{ }`.
- Variables are declared by giving the type, followed by one or more type identifiers. Ie, `int x, y ;` declares two variables `x` and `y` that can hold an integer
- Channels are declared by prepending a type declaration with the keyword `chan`, so `chan int s, t ;` declares two channels, each channel being capable of transmitting an integer. The input ends of these channels are denoted `s?` and `t?`, the output ends of these channels are denoted `s!` and `t!`.
- Port variables (ends of channels) are of the type '`?`*type*'. So the statement `?int a,b;` declares two ports, `a?` and `b?` each being the input end of a channel of integers.
- Communication is denoted using juxtaposition, ie, to send the number `3` over port `t!` we would write `t!3`, and to receive data over a port `t?` into a variable `x`, we would write `t?x`.

2 Semantics of Mobility

Mobility is the requirement to move an activity and/or information to a remote processor. There are three ways to encapsulate this operation in a programming language:

Copying semantics Copying semantics copies an object or activity from the source processor to the destination processor. This is also known as a cloning operation.

Moving semantics With moving semantics we destroy the original object once it has been copied to the remote processor. This is known as a teleport operation.

Borrowing semantics Borrowing semantics constraints the moving semantics in a lexical scope, so that what is moved out will always come back.

Below we will discuss the three classes and give some examples.

2.1 Copying semantics

With copying semantics each object or process is cloned, where the clone resides on another processor in the system. The cloning process has long been used as the basis of networking protocols, for it neatly fits the hardware model. When data is sent from one machine to the other, a remote copy is created, while the original data is not affected.

Cloning data is fine for simple data such as numbers, but becomes rather awkward when looking at large data structures (for example video streams), highly structured data (such as graphs) or communication channels. Copying large data structures is a waste of time. In many cases when a data structure is to be transferred to another process, the local process no longer needs a copy. For example, instead of copying frames of a video stream it may be cheaper to to pass on each frame of the stream (especially if the communication mechanism happens to allow shared memory). In addition, complex data structures are difficult to marshal, hence it is undesirable to copy them. Channels in the Occam sense are point to point; if we copy one end of a channel, we end up with a many-to-one channel.

2.2 Moving semantics

Moving semantics is based on the idea of destroying the original at the same time as creating the remote copy. In other words, there is no remote *copy*, instead we move the data bit by bit to the remote processor, destroying the original while building the remote one. This is also known as a teleport operation.

The advantage of moving instead of copying is that there is only one copy at any time, hence the compiler and run time support system can optimise data transfer. For example, in the layers of a networking protocol, one can move the data downstream with zero copying by definition. On a shared memory machine, one can move large or complicated data structures by simply handing over a pointer. NIL [1, 2] uses this principle, and it is the paradigm used by Barnes and Welch [6].

2.3 Borrowing semantics

Borrowing semantics is based on the same idea as moving semantics, except that there is a guarantee that the data is returned at the end of the transaction. Therefore, any data item that goes one way will come back when the operation is finished.

The advantage of borrowing is that we can avoid management of "empty" variables and processes. If we consider a variable as a box that can hold a value, then with moving semantics the language will have to manage which variables contain a value, and which variables do not contain a value (because the value has been moved to another processor). With borrowing semantics we can hide this issue, because the scope of the language can prevent access to the variable in the scope where the value is borrowed by another processor.

3 Mobile code

If we apply copying semantics for migrating code, we have a side effect that in addition to migration, we have created an extra thread of execution. So instead of migrating the process, we would have migrated a copy of the process, while the old one is still running (combining the unix `fork()` operation with a migration). More interesting are the moving and borrowing semantics. The moving semantics would move the current process away. For example:

```
int a, b, c ;
a = 1 ;
move to ... ;
b = a + 1 ;
c = b + a ;
```

would move the execution to another processor (specified on the dots) on execution of the move-to statement between the assignments to `a` and `b`. The effect could be be dramatic, in that when a procedure returns, the computation may have teleported to a different processor. Also, we will have to define the semantics of a parallel statement where one or more processes move.

Instead, we have chosen borrowing semantics. In Icarus, process migration is supported with the *on*-statement. An on-statement executes a statement on another processor. Semantically the *on* is a no-operation. The result of remote execution is exactly equivalent to local execution. In particular all communication capabilities continue as if no migration is taking place, so the statements:

```
int a, b, c ;
a = 1 ;
on( ... ) {
  b = a + 1 ;
}
c = b + a ;
```

will result in the value of 3 in `c`. The on-statement has clean semantics thanks to scoping of the statements to be executed remotely. From an operational viewpoint all variables that are used in the scope of the on-statement are marshaled and sent to the remote processor, whereupon the computation continues remotely; when finished, all the data is marshaled back, and the original code resumes. All operations are synchronous, and even if the data to be transmitted is complex (arrays, channels), they always come back. Resource allocation and reclamation is exactly as simple as it was without the on-statement.

In Icarus, the destination of the on-statement is specified as a port. The on-statement continues execution on the processor where the corresponding port resides. This way, the code is guaranteed to have zero-latency communication on one particular channel, maybe at the expense of increased communication costs over other channels.

Because on-statements are semantically invisible (they only have a performance effect), they can be inserted during or after program development, in order to migrate processes in such a way that latency is minimised. An example program to transfer video is:

```
encode( chan frame input, chan frame output,
        chan pixels screen ) {
  chan byte transmit ;
  par {
    capture( input ) ||
    encode( input, transmit ) ||
    on( screen ) decode( transmit, output ) ||
    on( screen ) display( output, screen )
  }
}
```

This assumes that we have four procedures `capture` (which captures a video-camera signal into a stream of frames), `encode` (which compresses a stream of frames into a stream of bytes), `decode` (which uncompresses a stream of bytes), and `display` (which views a stream of frames). The last two processes are executed on the machine with the display; so the channel `transmit`, used to transmit the compressed stream, spans the transmitting and receiving processor.

The on-statement can be used to, for example, migrate decryption algorithms, decompression algorithms, or user interfaces (web browsers, games) to the client.

4 Mobile communication

Even though the semantics of the on-statement are trivial, its implementation does require some work. In particular, communication capabilities have to be transferred with the process being transferred. In order to implement this we have implemented mobile channel protocols, which allow both channel-ends to be migrated independently at minimal costs [3, 4]. This is similar to the pi-calculus [5], except that our channels are guaranteed to be one-to-one connections.

Now that we have a protocol available that marshals ports from one processor to another, we can make that protocol visible to programmers, and allow them to migrate ports around. In the implementation of Icarus we have a primitive available that migrates a port from one processor to another. There are three ways to encapsulate this in Icarus (Copying semantics are a non-starter, for they would create many-to-many channels):

Borrow operation With the borrow operation we temporarily hand a port over from one process to the next. This process can be scoped lexically.

Move and view Channels as first class citizens This goes a step further than the borrow operation in that we allow channel ends to migrate. The semantics are slightly messy.

Move and view Ports as first class citizens This reduces the channel to be an abstraction which is connected to two ports, and makes the semantics clearer, at the expense of giving almost complete freedom to the programmer.

4.1 Borrow operation

The borrow operation is similar to the Occam-3 call channel, and a ADA rendez vous. We allow channels to be communicated on a borrowing basis, that is, the channel is passed on to another process temporarily, because at the end of the scope of the borrow operation, the channel will move back. In the following example we have three processes, we assume that there are channels `s` and `t` connecting processes 1 and 2, and processes 1 and 3. Process 1 sends the end of a channel `a` to processes 2 and 3, and allows processes 2 and 3 to communicate data:

process 1:

```
{
  chan int a ;          /* 1 */
  call( a? to t! ) {    /* 2 */
    call( a! to s! ) { /* 3 */
      ;
    }                   /* 4 */
  }                     /* 5 */
}
```

process 2:

```
  accept( ?int x from t? ) {
    int p, q ;
    x? p ;
    x? q ;
    p = p + q ;
  }
```

process 3:

```
  accept( !int z from s? ) {
    z! 1 ;
    z! 2 ;
  }
```

The `call` syntax is used to send data over a channel in a borrowing fashion: `call` x `to` y means that the process on the other side of y can borrow port x for the scope of the `call`; `accept` x `from` y means that this process will accept a port x from the other side of y for the scope of the `accept`. The call and accept are synchronised in that the call and accept will wait for each other on entry, and on exit of their scope.

The advantage of this system is that mobility of channels is constrained in that every channel that is sent shall be returned; hence, no communication capabilities are ever invented or destroyed. In the scope of the call statement, the channel that migrates is out of scope (because it is on the callee's side), while the channel is only in scope in the accept statement of the callee.

So, at point `/* 1 */` both `a?` and `a!` are part of Process 1, at point `/* 2 */`, `a?` is no longer in scope (it has been transferred over `s`), at point `/* 3 */` both `a?` and `a!` are out of scope, at point `/* 4 */`, `a!` comes back in scope, and finally at point `/* 5 */` both ends are back in scope, only to disappear from scope on the next line.

In the example above, there are still ordinary input and output operations. This mixture of borrowing and non-borrowing semantics may be confusing, and this can be simplified so that only borrowing semantics are used. Below, we repeat the example using only borrowing

semantics, where we have used the conventional !? syntax, but with a scope in which the data is borrowed/lent:

process 1:

```
chan int a ;
t! a? {
  s! a! {
    ;
  }
}
```

process 2:

```
t? ?int x {
  x ? int p {
    x ? int q {
      p = p + q ;
    }
  }
}
```

process 3:

```
s? !int z {
  z! 1 {
    z! 2 ;
  }
}
```

On reception of a value from a channel one must declare a variable to hold the result. The contents of that variable go out of scope on finishing the scope. Note the difference between sending data subsequently, or scoped:

```
chan int a ;
t! a? {
  s! a! {
  }
}
```

```
chan int a ;
t! a? {
}
s! a! {
}
```

The right hand side will complete the synchronisation on `t!` before starting the output on `s!`, whereas the left hand side will output on `s!` before completing the synchronisation on `t!`.

The disadvantage of the borrowing scheme is that many mobile issues cannot be expressed using this scheme. In particular, we cannot give two processes each one end of a channel (so that they can communicate), and then let them use it for as long as they want without the original process's involvement. For example, a way to model a mobile-phone switchboard would be to continuously create channels, and pass the ends to mobile phones who can communicate without intervention of the switchboard. Using the syntax above, the switchboard must at any time wait for all the channels that are sent out to return.

4.2 Channels as first class citizens

A way around the inflexibility of borrowing is to make channels first class citizens. In effect, we lift the primitives for sending and receiving channels that we developed for the on-statement, and we make those visible at language level. We need some additional notation to complete the process. Our three processes above would be formulated as follows:

process 1:

```
{
  chan int a ;  /* 1 */
  t! a? ;       /* 2 */
  s! a! ;       /* 3 */
}
```

process 2:

```
{
  ?int x ;
  int p, q ;   /* 4 */
  t? x? ;      /* 5 */
  x ? p ;      /* 6 */
  x ? q ;
  p = p + q ;
}
```

process 3:

```
{
  !int z ;     /* 7 */
  s? z! ;      /* 8 */
  z! 1 ;       /* 9 */
  z! 2 ;
}
```

The first process takes the two ends of the channel `a` and sends them over s and t; the second process declares a *port* `x?`. This port variable is read from `t?` and subsequently `p` and `q` are read from `x?`. Similarly, the third process receives the other end of `a` into `z!`, and sends data over it.

The difference with the previous part is that the scope and the extent (life-time) of channels have been detached. The channel `a` of process 1 goes out of scope on the fifth line, but the channel is still alive, for it is used by processes 2 and 3. The life time of the channel ends at the end of process 2 and three, when the ports `x?` and `z!` go out of scope.

To get this scheme to work, we must define what happens when we input and output ports over channels, and what the meaning of port variables is.

Inputting and outputting ports over channels involves *moving* the port from one end to the other. It is moved rather than copied, for copying would result in a many-to-one or one-to-many channel. Because the channel is moved, the process outputting the port effectively loses the port, whereas the process inputting the port gains the port. This means that at each point in the code, a port can be in one of two states: connected and unconnected. In the example above:

- At point `/* 1 */` channel a consists of two ports, both connected.

- At point `/* 2 */` channel a consists of two ports, the input port is no longer connected.
- At point `/* 3 */` channel a consists of two ports, neither connected.
- At point `/* 4 */` x is an unconnected port.
- At point `/* 5 */` and `/* 6 */` x is a connected port.
- At point `/* 7 */` z is an unconnected port.
- At point `/* 8 */` and `/* 9 */` z is a connected port.

we can either defer checking connected of ports to run time, or we can perform compile time checks of connectedness. We prefer to define connectedness as a static property of the program, ie, at any part of the code it must be statically determined that a port is connected or not, and it is only legal to have input and output statement on connected ports.

This scheme is far less constrained that the previous scheme; but it has lost the semantic simplicity of ports always being connected and lexically scoped. In addition, we have created a situation in process 1 where channel a has only one connected port.

An advantage over the previous scheme is that one can define structured data types (records, arrays, variant records) containing ports and that these can be transported over channels.

This process is very similar to the NIL language by Strom [1, 2].

4.3 Ports as first class citizens

The final step is to do away with channels as an entity that can be declared, and make the port the primary data-type. In this case, we replace the channel declaration with two port declarations, and we have an operator to create a channel connection between two port variables:

process 1:

```
{
  int a? ; int a! ;
  a! => a? ;
  t! a? ;
  s! a! ;
}
```

The code of processes 2 and 3 stays the same; note that we declare the two ports, and use the `=>` operator to create a connection. This makes it explicit that we can create as many channels as we like. The `=>` operator requires two ports of matching types, both unconnected, and will create a channel. The close operator `^` will close a port and unconnect it. A port is implicitly unconnected when it leaves its scope. Unconnecting a port only succeeds when the other side of the channel unconnects.

This goes one step further, and exposes the channel creation/destruction operators to the programmer.

4.4 Comparison

We have so far implemented the second scheme, and have performed some brief experiments with the third scheme. We think that the first scheme is superior in simplicity, but that it is not rich enough to implement highly mobile applications.

5 I/O, user I/O, and URL ports

If we choose one of the latter two mechanisms to transport ports, we can use an elegant mechanism to connect the user to the program, and to embed an application program into the Internet.

5.1 URL ports

Channels as presented above will once created in a program stay within that program. In order to connect unrelated programs (clients, servers, users), Icarus supports “URL ports”. These are ports that are identified by means of a Uniform Resource Locator. A URL port is a string (URL) which is used in place of a port in an input or output operation. As an example the following program will connect two unrelated ports with a channel (note that in this example the processes reside in the same program, but the example also works when the processes are unrelated):

```
proc process_server( ?int x ) {
  "/url" ? x? ;
}
proc process_client( ?int x ) {
  "/url" ! x? ;
}
par {
  {
    ?int x ;
    process_server( x? ) ;
    x ? a ;
    x ? a ;
  }
||{
    ?int z ;
    !int y ;
    y! => z? ;
    process_client( z? ) ;
    y!1 ;
    y!42 ;
  }
}
```

The first process creates an input port `x`. The port is then input from a channel identified by a the string `"/url"`. The other process creates a channel, `y`, and sends the input port of that channel to the channel identified by the string `"/url"`. The two strings match, the types of the channels match, therefore the communication succeeds, and process one has one port of the channel, while process two still has the other end of the channel.

The URLs above are local URLs. When they are fully qualified, they have a prefix `dcp`: `"dcp://host.domain.name/path"`. The `dcp` prefix stands for “Daedalus Channel Protocol”, which is the underlying protocol.

URL ports provide, like ordinary ports, a point to point communication medium. However, URL ports exist for only one transaction. The input from a URL port will advertise its presence, with type information, and wait for the other side. The outputting process will search for an inputting process, connect, one message will be transferred, and the channel will then be shut down.

Multiple URL ports may exist at the same time. If multiple URL ports have the same name, they are handled one after the other in a non deterministic order. This allows a server to serve multiple clients.

Usually one outputs a port over a URL channel, thereby connecting the two processes via an ordinary channel. Because the current version of Icarus does not support tuples, not more than a single port can be output over a URL channel. This means that in order to setup two way communication, two URL communications are necessary.

5.2 User I/O

User communication can be performed over URL ports in a neat and efficient way. There are, at present, 2 default ports to connect users. The first default port is one to receive information from a client. Keyboard events and mouse-clicks are input over this channel. The other default port is an output channel, to render graphics and text on the client screen. The types of these ports are `?int` and `![6]int` respectively. A group of functions is defined to handle input and output over those ports.

The user-I/O mechanism defines a user as a client (this is typically a web browser), and any program that is running as a server. The server can wait for a connection from the user by inputting on a URL port. On this URL port, the program can obtain the two user I/O channels, which can subsequently be used to process I/O.

```
proc main() {
  ?int event ;
  ![6]int graphics ;
  int z, x, c, y, but ;

  par {
    "/rectangles" ? event ;
  ||
    "/rectangles" ? graphics ;
  }
  on( graphics ) {
    while( true ) {
      event ? z ;
      interpretevent( event, z, c, x, y, but ) ;
      if( c == 0 ) {
        drawrect( graphics,x-5,y-5,x+5,y+5,64*but ) ;
      }
    }
  }
  userclose( graphics, event ) ;
}
```

The first two lines of the program will wait for the user input and output channels. After both have been connected (both statements will terminate), the rest of the program will migrate to the user and perform the following activities:

- Wait for an event on the user channel
- Interpret the event to determine whether it was a mouse click or a keyboard event.

- If it is a mouse click, then draw a rectangle around the mouse in a colour specified by the button that was used.

The functions `interpretevent` and `drawrect` are two standard library functions that decode / encode the user protocol. Note that the on-statement is used here to move the code performing the user interface to the user. In a typical application, all user I/O code will migrate to the user, providing interactive I/O to the user, at the expense of a high latency channel between server and user-interface.

6 Current status and conclusions

At present we have an implementation of Icarus which supports the on-statement and the third form of port migration. In this paper we have only discussed the Icarus approach to *migration* of ports and processes. Because ports are first class citizens, a variable can be assigned another port, just like any a variable of any other type; this uses the same moving semantics.

The Icarus compiler compiles Icarus to byte-code (Daedalus), which can in turn be compiled into C (at the expense of portability), or it can be compiled just-in-time, again via C. The run time support library is small, the protocol to move ports around is very simple because channels are always one-to-one. If many-to-many channels are allowed, we would have to implement channel clearing operations prior to migration. As it stands, we only move one or two ends of a synchronous one-to-one channel, so exactly one other party is involved in the migration, this other party can in the worst case be in the process of outputting a single data item.

The compiler has been used by two classes of students to write mobile code. The learning curve on using the on-statement properly was negligible. Only few students used first class channels. We are currently investigating which of the channel migration mechanisms is the cleanest, and which is the easiest to program. We are going to perform these experiments in the context of our wearables programme; which requires a highly mobile programming language.

References

[1] Rob Strom and Shaula Yemini. The NIL Distributed Systems Programming Language: A Status Report. *SIGPLAN notices*, 20(5):36–44, May 1985.

[2] Rob Strom and Shaula Yemini. NIL: An Integrated Language and System for Distrbiuted Programming. pages 73–82, 1983.

[3] Henk L. Muller and David May. A Simple Protocol to Communicate Channels over Channels. In *EURO-PAR '98 Parallel Processing, LNCS 1470*, pages 591–600, Southampton, UK, September 1998. Springer Verlag.

[4] David May, Henk Muller, and Shondip Sen. Hardware Migratable Channels. In *Euro-Par 2000 Parallel Processing*, pages 545–549, Munchen, September 2000. Springer Verlag, LNCS 1900.

[5] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, I. *Information and Computation*, 100(1):1–40, September 1992.

[6] F.R.M. Barnes and P.H. Welch. Mobile Data, Dynamic Allocation and Zero Aliasing: an occam Experiment. In *Communicating Process Architectures 2001*, Sep 2001.