Bypassing of channels at Process Creation Time

Ricardo Peña Clara Segura

December 15, 1998

1 Compile time analysis

Full Eden is coded into Haskell at parsing time and then translated into Core by the rest of GCH front-end phases (type checking and desugaring). From there, occurrences of `#` and `process` functions are detected and can be converted into Extended Core, in which process abstractions and instantiations are explicit and each single channel is an explicit variable. The following Eden program:

```
pipe :: [Process a a] -> Process a a
pipe [p]    = p
pipe (p:ps) = process x -> pipe  ps # (p # x)
y = pipe [p1,p2,p3] # z
```

will be translated into the following Extended Core one:

```
pipe = \ ps -> case ps of
                [p]  -> p
                p:pp -> process x ->
                           let x´= # p x
                               p´= pipe pp
                               y = # p´ x'
                           in  y
pa  = p1:pb
pb  = p2:pc
pc  = p3:[]
myPipe = pipe pa
y = # myPipe z
```

Process instantiations are floated as much as possible and, because of this, some `let`'s may be joined together into bigger `letrec´`s. A program analysis is done on the representation resulting from this. After that, the result is coded again into Core. Of course, Core2Core transformations should be delayed until this process has been completed.

At first sigth, the analysis consists of a variant of sharing analysis. In sharing analysis the underlying abstract domain is the set of subsets of $\{0, 1, Many\}$ with $\subseteq$ as the ordering relation $\sqsubseteq$. Here, we seek for exactly two occurrences of a channel variable, one as inport and one as outport. After the analysis we get a transformed annotated Core program in Ulrike´s style. In our example:

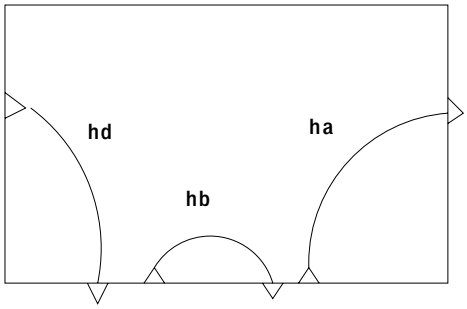

Figure 1: Graphical conventions

```
pipe = \ ps -> case ps of
               [p]  -> p
               p:pp -> let hd = createBypass ()
                           ha = createBypass ()
                       in
                       process (\ x ->
                         let hb = createBypass ()
                             x´= ## p (sendTup x) ([1 -> hb],[[1 -> hd])
                              p´= pipe pp
                              y = ## p´ (sendTup x') ([1 -> ha],[1 -> hb])
                         in  y) ([1 -> hd],[1 -> ha])
pa  = p1:pb
pb  = p2:pc
pc  = p3:[]
myPipe = pipe pa
y = ## myPipe (sendTup z) ([],[])
```

In graphical terms, the annotations associated to the process abstraction can be depicted as in Figure 1. We use the following graphical conventions throughout the paper: in the left hand side of the box, we draw the inports belonging to the process abstraction external interface and, similarily, we show the external outports on the right hand side. Inports and outports relating the process abstraction to its children are shown on the bottom side of the box. We will call "handle between brothers" handles such as `hb`, "ascendant handles" the ones such as `ha` and "descendant handles" the ones such as `hd`.

2 The bypassing protocol

2.1 Bypassing between brothers

One of the cases we can find in bypassing of channels arises when the parent process creates several children in a pipeline fashion. A simple example of this is the following:

```
p0:...
letrec
       x = p2 # y
       y = p1 # inp
```

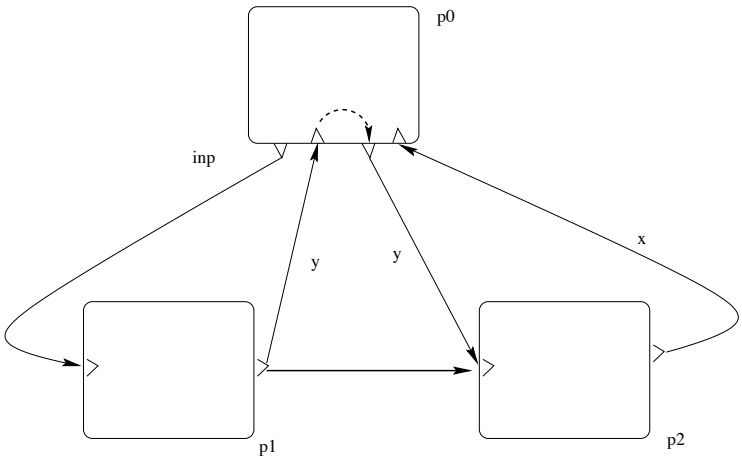

Figure 2: Bypassing between brothers

```
in  ...
```

This program will be translated into the following annotated Core program in Ulrike's style:

```
p0:...
letrec
      hb = createBypass ()
      x = ## p2 (sendTup y)   ([], [1 -> hb])
      y = ## p1 (sendTup inp) ([1 -> hb], [])
in ...
```

The current topology created for this situation is not the desired one, because one of the children produces a result sent to the parent and the parent only forwards this result to the following child and so on, see Figure 2. The desired topology, shown in thicker lines, connects directly the children.

2.2 Bypassing between generations

Another situation in which bypassing of channels would be useful arises when the input of a process coming from a remote outport is merely forwarded to its child or the output of a child is directly forwarded as an output of its parent. Examples of these two possibilities are the following:

```
(1)p0:...
    letrec
          x = p # y
          p = process y'  ->  f h
              where
                   h = q # y'
          q = process y'' ->  g i
              where
                   i = r # y''
```

```
    in ...

(2)p0:...
    letrec
           x = p # y
           p = process y'  -> h
               where
                    h = q # (f y')
           q = process y'' -> i
               where
                    i= r # (g y'')
    in ...
```

In the first example, see Figure 3, the output `y` sent by the parent `p0`, is forwarded along several descendants downwards. The desired topology, shown in thicker lines, connects directly the outport in the parent `p0` with the final destination in `r`.

In the second example, see Figure 4, the ouput produced by `r`, is forwarded along several ascendants upwards. The desired topology, connects directly the inport in the parent `p0` with the final origin in `r`.

This programs will be translated into the following annotated Core programs in Ulrike's style:

```
(1) p0:...
     letrec
           x  = ## p (sendTup y) ([],[])
           p = letrec
                  hd1 = createBypass ()
               in process (\y' ->
                           letrec h = ## q (sendTup y') ([], [1 -> hd1])
                           in (sendTup (f h))
                          ) ([1 -> hd1], [])

           q = letrec
                  hd2 = createBypass ()
               in process (\y'' ->
                           letrec i = ## r (sendTup y'') ([], [1 -> hd2])
                           in (sendTup (g i))
                          ) ([1 -> hd2], [])


    in  ...

(2)p0:...
    letrec
           x = ## p (sendTup y) ([], [])
           p = letrec
                  ha1 = createBypass ()
               in process (\y' ->
                           letrec h = ## q (sendTup (f y')) ([1 -> ha1], [])
                           in (sendTup h)
```

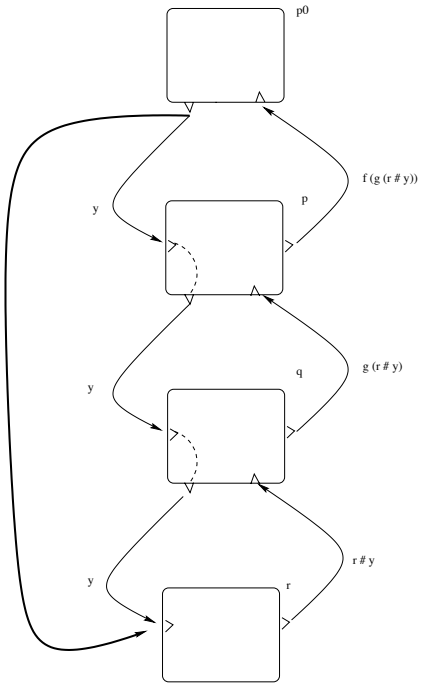

Figure 3: Bypassing between generations I

```
                    ) ([], [1 -> ha1])


        q = letrec
                  ha2 = createBypass ()
            in process (\y'' ->
                        letrec i = ## r (sendTup (g y'')) ([1 -> ha2], [])
                        in (sendTup i)
                       ) ([], [1 -> ha2])

    in  ...
```

In general, this situation may be recursive, so that a process' outport may be connected to a remote descendant's inport, or respectively a process' outport may be connected to a remote ascendant's inport, see Section 1.

2.3 General bypassing

In a more general situation a combination of them may happen:

```
p0:...
   letrec
         x = p2 # y
         y = p1 # inp
         p2 = process i -> (f h)
```

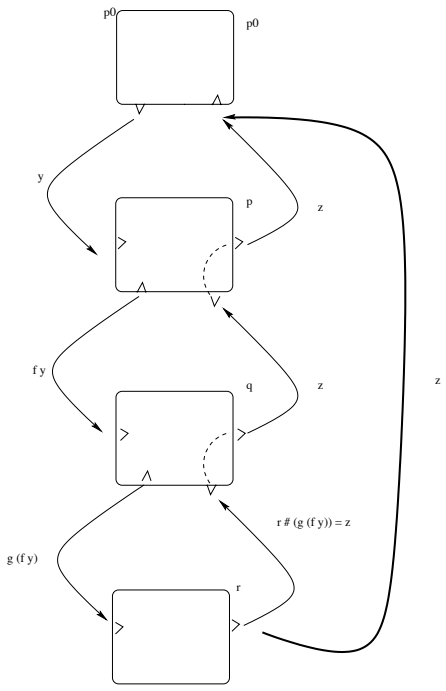

Figure 4: Bypassing between generations II

```
            where
                 h = p3 # i
in ...
```

In this example, see Figure 5, we have a combination of bypassing between brothers and generations. The desired topology connects directly the outport of `p1` with the inport of `p3`, as `p0` forwards the output produced by `p1` to `p2` and `p2` forwards it again to `p3`.

This programs will be translated into the following annotated Core programs in Ulrike's style:

```
(1) p0:...
    letrec
          hb = createBypass ()
          x = ## p2 (sendTup y)   ([], [1 -> hb])
          y = ## p1 (sendTup inp) ([1 -> hb], [])
          p2 = letrec
                    hd = createBypass ()
               in process (\i ->
                           letrec h = ## q (sendTup i) ([], [1 -> hd])
                           in (sendTup (f h))
                          ) ([1 -> hd], [])

    in
     ...
```

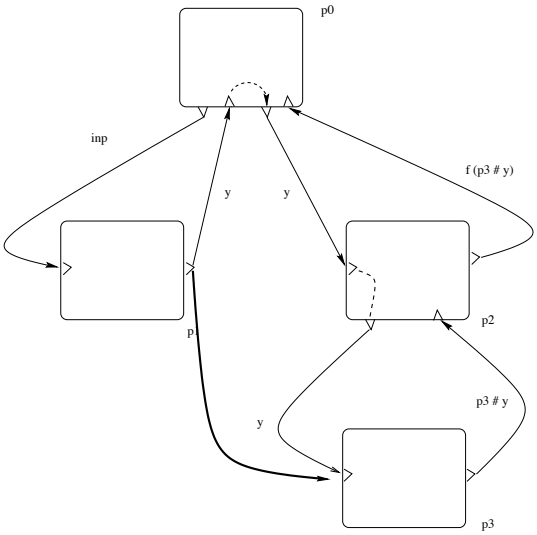

Figure 5: General bypassing situation

2.4 The messages

In the general situation, it is necessary to detect the final origin and destination of the data to be sent from one process to another. The whole hierachy is generated from the instantiation of a process `p0`, which we call the grandparent.

Each process in the hierarchy will send acknowledgement messages to its parent and creation messages to its children. The ACK message informs the parent about the inports and outports of its child, and the CREATE-PROCESS message informs the child about the inports and outports in the parent.

But in a bypassing situation, some of the inports and outports will not be created. The bypassings are identified by handles, so these handles will be propagated up and down in the ack and creation messages.

So in a creation message:

$$CREATE - PROCESS(idp_{parent}, p, o_{tso}, i'_1, \ldots, i'_k, o'_1, \ldots, o'_n)$$

and in an acknowledgement message:

$$ACK(idp_{child}, o_{tso}, i'_1, \ldots, i'_k, o'_1, \ldots, o'_n, i_1, \ldots, i_n, o_1, \ldots, o_k)$$

the i'_j and o'_l may be of the form:

- $(PE, port)$ if they are actual inports/outports
- $(PE, handle)$ if they correspond to a handle of type hb (we will see it is not necessary to send ha or hd handles)

In code generation, see section 3, it will be necessary another component, f, a boolean flag indicating if the destination/origin comes directly from the father ($f = True$) or it has been propagated trough intermediate processors ($f = False$). This will be necessary to optimize the number of messages, see section 2.6.

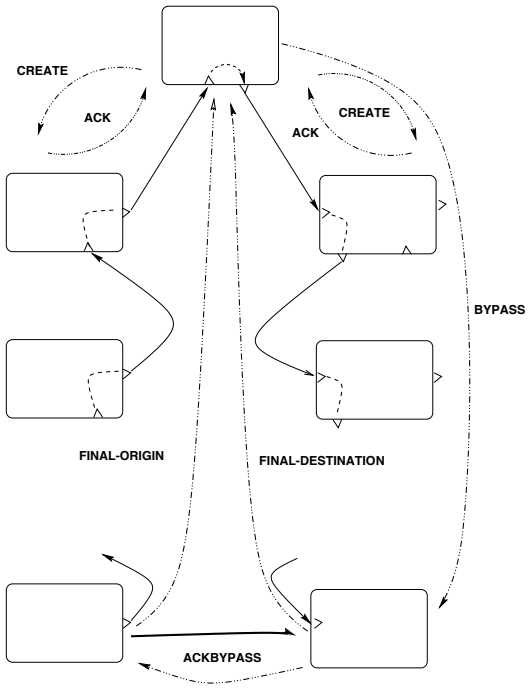

Figure 6: Bypassing protocol (1)

The i_l and o_j are just pairs (PE, x) where x may be a real port or a ?, indicating that the child's inport/outport corresponds to an hd/ha handle.

With this information the bypassing table is increasingly updated, see Section 3. When a descendant is detected as a final origin or destination (empty bypassing lists) it communicates the grandparent this information to get the bypassing table updated.

When the grandparent is provided with the whole information about the outport and the inport to be connected directly, it informs about such information to the involved processes and the desired connection is stablished.

So we need some new messages in the protocol:

- A FINAL-ORIGIN message from the final origin in the bypassing to the grandparent p_0, which updates properly the bypassing table.
- A FINAL-DESTINATION message from the final destination in the bypassing to the grandparent, which also updates the bypassing table.
- A BYPASS message to the final receiver informing about the PE and the outport from which it will receive the data.
- An ACKBYPASS message to the final sender informing about the PE and the inport to which it has to send the data.

2.5 The protocol

The sequence on messages is then the following (see Figures 6 and 7):

- In each level there is a creation message for each of the children followed in sequence by the corresponding ACK message. We don't impose any sequentiality between the creation

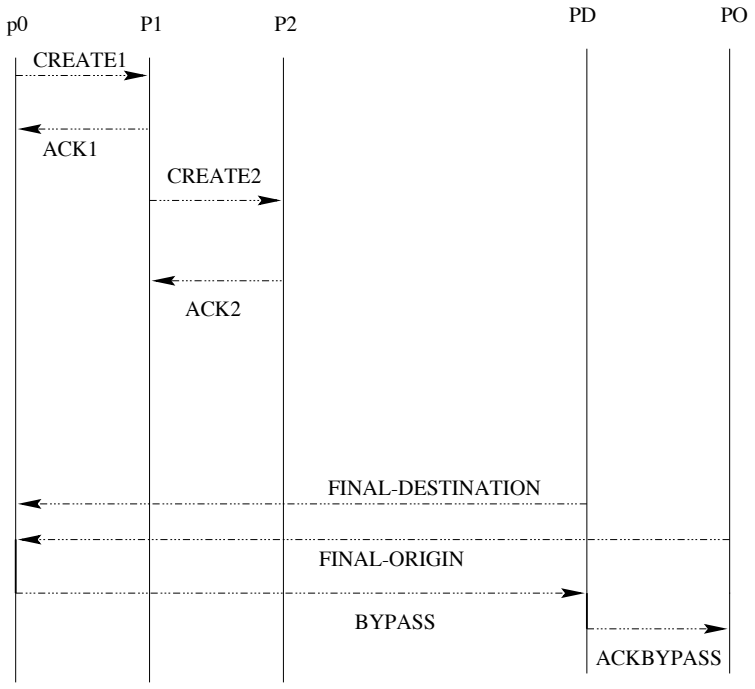

Figure 7: Bypassing protocol (2)

of brothers as we want the maximum parallelism and also to cope with cyclic connections between children.

- When all the connections of a process with respect to its parent are stablished, then it can begin creating its children. Therefore we impose a sequentiality between the creation of processes in different levels.

- When a descendant is detected as a final origin (empty inport bypassing list) it sends a FINAL-ORIGIN message to the grandparent:

$$FINAL-ORIGIN((PE_{sender}, outport_{sender}), (PE_{grandparent}, x))$$

  where:

  - $(PE_{sender}, outport_{sender})$ specifies the final origin in the bypassing
  - x may be a real inport or a handle hb in the grandparent. In the last case the corresponding entry in the bypassing table will be updated with the final origin information.

- When a descendant is detected as a final destination (empty outport bypassing list) it sends a FINAL-DESTINATION message to the grandparent:

$$FINAL-DESTINATION((PE_{receiver}, inport_{receiver}), (PE_{grandparent}, x))$$

  where:

  - $(PE_{receiver}, inport_{receiver})$ specifies the final destination in the bypassing

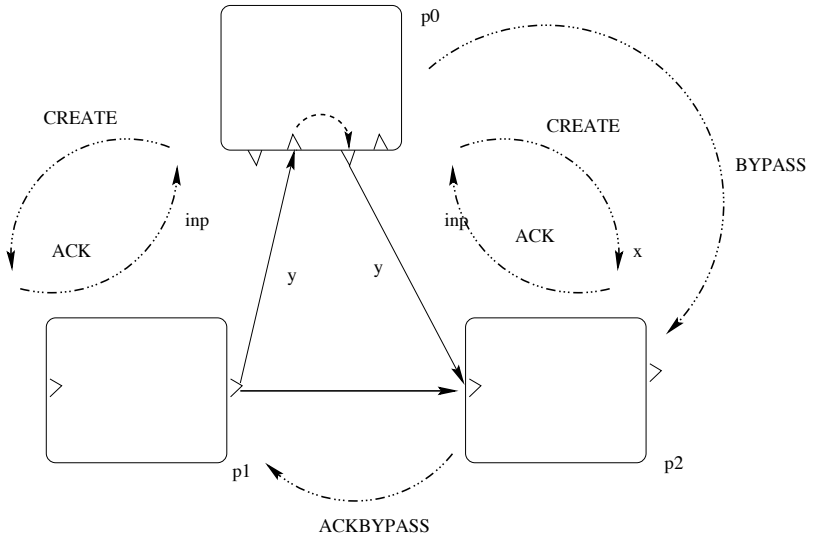

Figure 8: Symplified bypassing protocol (1)

- x may be a real outport or a handle hb in the grandparent. In the last case the corresponding entry in the bypassing table will be updated with the final destination information.

There does not exist any sequentiality between the $FINAL-ORIGIN$ and $FINAL-DESTINATION$ messages.

- When the grandparent receives both messages, it sends a BYPASS message to the final receiver:

$$BYPASS((PE_{sender}, outport_{sender}), (PE_{receiver}, inport_{receiver}))$$

showing the receiver the connection that must be done.

- Then, the final receiver sends an ACKBYPASS message to the final sender:

$$ACKBYPASS((PE_{sender}, outport_{sender}), (PE_{receiver}, inport_{receiver}))$$

There exists a sequentiality between the BYPASS and the ACKBYPASS message. They have esentially the same form but the code to be executed is different when they are received, see code generation in Section 3. Should this not be done, the sender could begin sending data while the receiver is not yet prepared to receive them.

2.6 Special situations

The general bypassing protocol may be simplified in some particular cases, specifically in pure situations, i.e. only bypassing between brothers or between generations.

In case we have a pure bypassing between brothers, like in example of Section 2.1, the final origin and final destination are the brothers themselves, so if we detect this situation we could avoid the FINAL-ORIGIN and FINAL-DESTINATION messages, as the ACK messages are enough to obtain the whole information, see Figure 8

In the pure bypassing between generations we have two different cases:

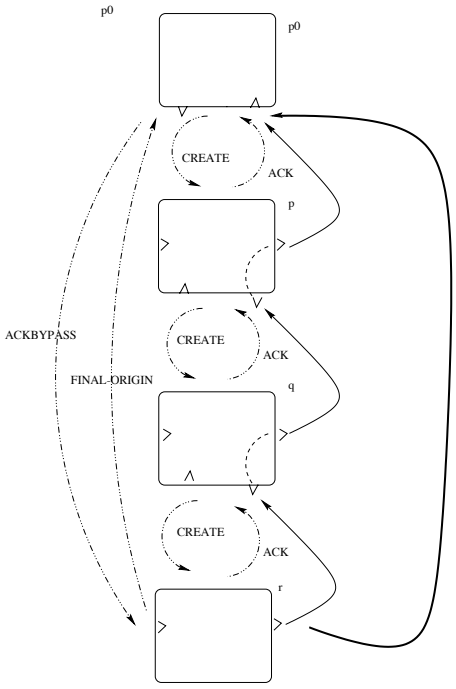

Figure 9: Symplified bypassing protocol (2)

- If we have only downwards bypassing, then the final origin is the grandparent itself, so we don't need to send a FINAL-ORIGIN message. Also, the BYPASS and ACKBYPASS messages are unnecessary (see Figure 10), as the inport connection is achieved in the final receiver when it is instantiated. The protocol is reduced then to the sending of a FINAL-DESTINATION message.

- If we have only upwards bypassing, then the final destination is the grandparent itself, so we don't need to send a FINAL-DESTINATION message and also the BYPASS message is unnecessary (see Figure 9). The protocol is reduced then to the sending of a FINAL-ORIGIN message followed by an ACKBYPASS message to the final sender.

2.7 A recursive example

This bypassing protocol works in the general cases. In the pipeline example in Section 1 the bypassing situation is a combination of several bypassing. The situation is shown in Figure 11 and the protocol in Figure 12.

3 Code generation

3.1 Conventions

- The acronym DBT will denote the *Dynamic Bypassing Table* located in the RTS associating every handle identifier to its respective origin and destination.

- When in downwards messages, a **destination** i_j is a triple (PE, f, x) where x can be either an inport or a handle and f is a boolean flag: $f = True$ means that the destination

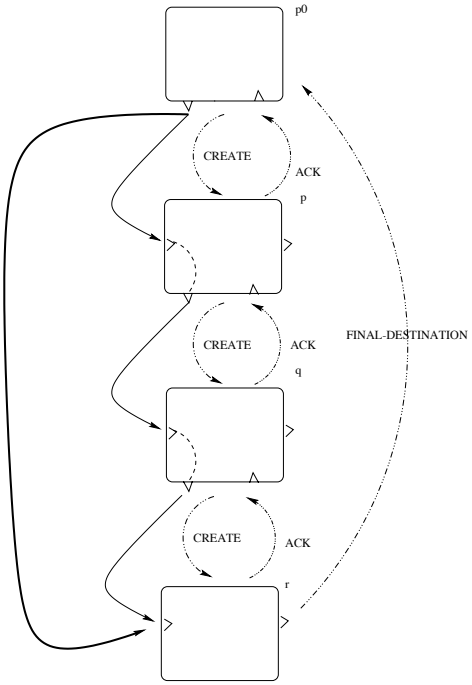

Figure 10: Symplified bypassing protocol (3)

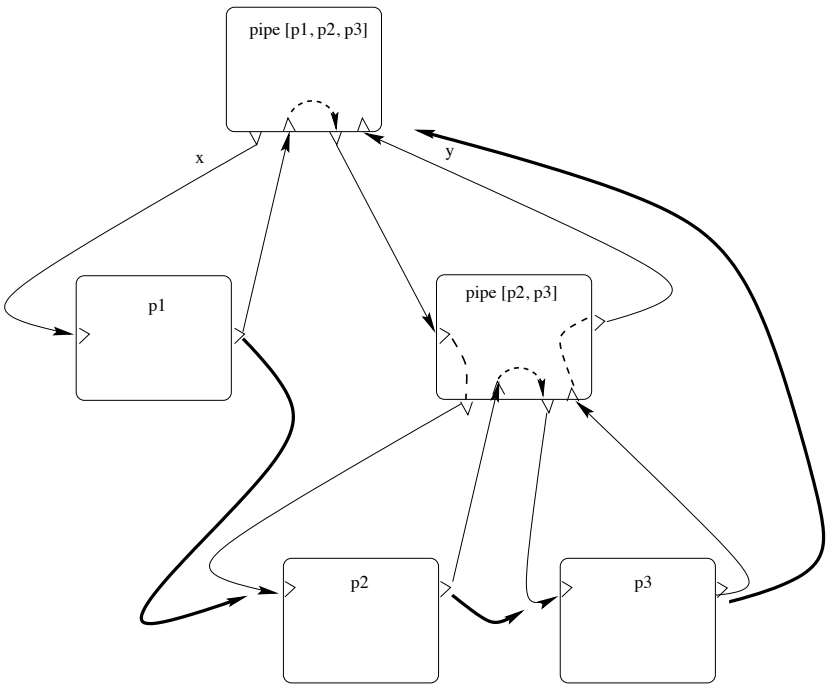

Figure 11: Pipe

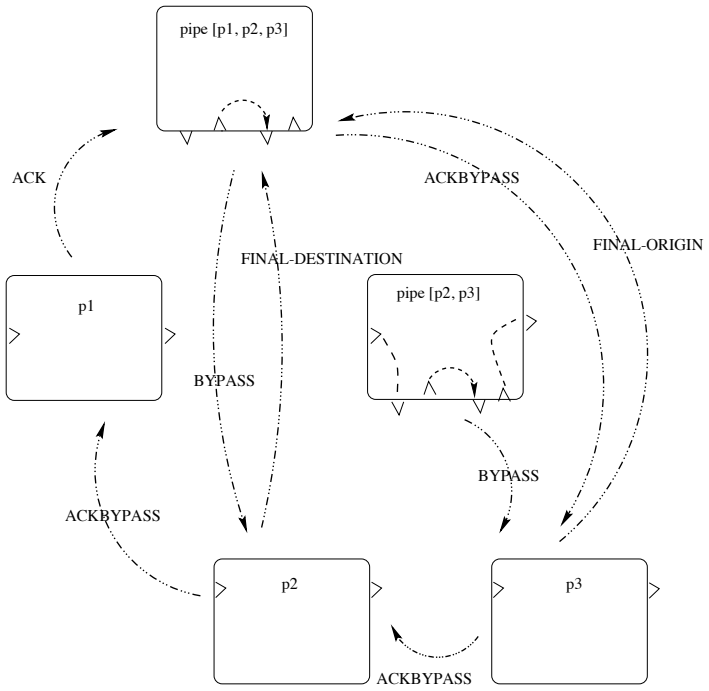

Figure 12: Pipe protocol

comes directly from the father, and $f = False$ means that it has been propagated through intermediate processors.

- When in downwards messages, an **origin** o_j is a triple (PE, f, x) where x can be either an outport or a handle and f is a boolean flag with the same meaning as in the precedent item.
- When in the DBT, both origins and destinations are represented as tuples (PE, x) where x can be either an actual outport/inport or a handle between brothers.
- $realInport(j, bi)$ gives $True$ if in the bypassing annotation bi input j is not annotated has having a handle.
- $annotatedIB(j, bi)$ gives $True$ if in the bypassing annotation bi input j is annotated has having a handle between brothers.
- $annotatedIA(j, bi)$ gives $True$ if in the bypassing annotation bi input j is annotated has having an ascendant handle.
- $annotatedID(j, bi)$ gives $True$ if in the bypassing annotation bi input j is annotated has having an descendant handle.
- $realOutport(j, bi)$ gives $True$ if in the bypassing annotation bi output j is not annotated has having a handle.
- $annotatedOB(j, bi)$ gives $True$ if in the bypassing annotation bi output j is annotated has having a handle between brothers.

- *annotatedOA*(j, bi) gives $True$ if in the bypassing annotation bi output j is annotated has having an ascendant handle.
- *annotatedOD*(j, bi) gives $True$ if in the bypassing annotation bi output j is annotated has having an descendant handle.
- $h \leftarrow$ *handleI*(j, bi) gives the handle associated to input j in the bypassing annotation bi.
- $h \leftarrow$ *handleO*(j, bi) gives the handle associated to output j in the bypassing annotation bi.
- *setDestination*$(h, (PE, x))$ registers in the DBT that handle h has x as destination. If h is between brothers, x can only be an actual outport. If h is ascendant, x can be either an actual outport or a handle between brothers.
- *setOrigin*$(h, (PE, x))$ registers in the DBT that handle h has x as origin. If h is between brothers, x can only be an actual inport. If h is descendant, x can be either an actual inport or a handle between brothers.
- $d \leftarrow$ *getDestination*(h) gets the destination of handle h from the DBT and delivers it in d. If it is undefined, returns the value ?.
- $d \leftarrow$ *getOrigin*(h) gets the origin of handle h from the DBT and delivers it in d. If it is undefined, returns the value ?.
- *disposeHandle*(x) deletes entry for handle x in the DBT.
- *outport*(x) / *inport*(x) / *handle*(x)/ *handleB*(x) gives $True$ if x is respectively an outport / inport / handle/ handle between brothers.

3.2 Process instantiation

We need to modify the code generation of the different constructions to treat bypassing, as some of the tasks usually achieved are now eliminated (see Figure 13). For instance, if a father input is associated to a handle, the corresponding inport and `queueMe` closure should not be created. Also, if a father output is associated to a handle, the corresponding outport is not created and the associated thread should not be created by the function `create?Thread` (see below). The new proposed code is:

$\#\ \ (v_1, v_2, k, n, bi) \stackrel{\text{def}}{=}$
for $j \in \{1..k\}$ **do** −− father's inports
$[\ realInport(j, bi) \rightarrow a_j \leftarrow createQM(); i'_j \leftarrow (ownPE(), True, createInport(a_j))$
$[]\ annotatedIB(j, bi) \rightarrow i'_j \leftarrow (ownPE(), True, handleI(j, bi))$
$[]\ annotatedIA(j, bi) \rightarrow (PE, x) \leftarrow getDestination(handleI(j, bi)); i'_j \leftarrow (PE, False, x)$
$]$
end for ;

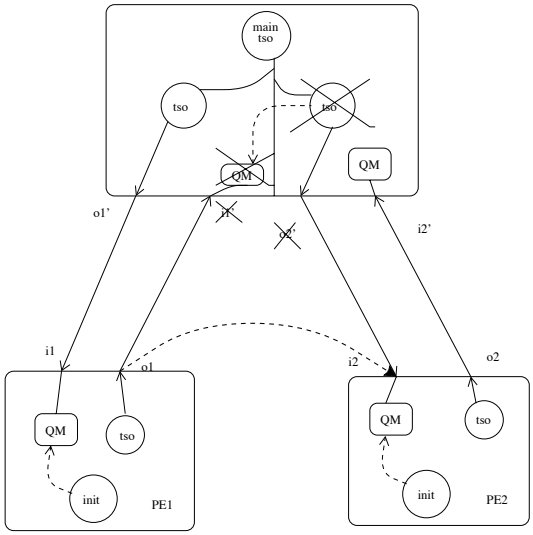

Figure 13: The changes

for $l \in \{1..n\}$ **do** −− father's outports
[$realOutport(l, bi) \rightarrow o'_l \leftarrow (ownPE(), True, createOutport(undefT))$
[] $annotatedOB(l, bi) \rightarrow o'_l \leftarrow (ownPE(), True, handleO(l, bi))$
[] $annotatedOD(l, bi) \rightarrow (PE, x) \leftarrow getOrigin(handleO(l, bi)); o'_l \leftarrow (PE, False, x)$
]
end for
$tso \leftarrow createThread(v_2, undefD)$;
$o_{tso} \leftarrow createOutport(tso)$;
$PE_{child} \leftarrow sendCreateProcess(v_1, o_{tso}, i'_1, \ldots, i'_k, o'_1, \ldots, o'_n)$;
$returnTuple(a_1, \ldots, a_k)$

3.3 Reception of CREATE-PROCESS at child's side

In this case we don't need to modify anything. When the message

$$CREATE - PROCESS(PE_{parent}, p, o_{tso}, i'_1, \ldots, i'_k, o'_1, \ldots, o'_n)$$

arrives, the closure p is unpacked and copied to the local heap. Let p' denote the address of the copied closure. Then the following actions are executed:

$tso \leftarrow createThread(p', undef)$;
$initStack(tso, PE_{parent}, o_{tso}, k, n, i'_1, \ldots, i'_k, o'_1, \ldots, o'_n)$;
$scheduleThread(tso)$

3.4 Process abstraction

In this code the first thing we do is to obtain parent's inports and outports from the stack, some of which, may be handles. Besides that, some child's inports and outports may be annotated as being respectively descendant/ ascendant handles. We must inform the parent of this circumstance in the ACK message. Besides that, we register in the DBT the origin/destination associated to the handle.

In case a child's inport/outport is a real one, we must send the FINAL-DESTINATION / FINAL-ORIGIN message to the PE having the corresponding origin/destination of the channel, except in the case that this origin/destination belongs to the parent process. When this happens, we save the FINAL-DESTINATION / FINAL-ORIGIN messages as the information is coded in the ACK message.

So, the new translation is the following:

$$
\begin{array}{ll}
process & (v,n,k,bi) \stackrel{\text{def}}{=} \\
& PE_{parent} \leftarrow popStack(); o_{tso} \leftarrow popStack(); \\
& \{i'_j \leftarrow popStack();\}_{j=1}^{k}; \{o'_l \leftarrow popStack();\}_{l=1}^{n} \\
& \textbf{for } j \in \{1..k\} \textbf{ do} \qquad\qquad -- \text{ child's outports} \\
& [\, realOutport(j,bi) \quad \rightarrow (PE,f,x) \leftarrow i'_j; \\
& \quad [\quad f \rightarrow o_j \leftarrow (ownPE(), createOutport(undefT)) \\
& \quad \Box\, \neg f \rightarrow o_j \leftarrow (ownPE(), createOutport(undefT)); sendFinalOrigin(PE,x,o_j) \\
& \quad] \\
& \Box\, annotatedOA(j,bi) \rightarrow (PE,f,x) \leftarrow i'_j; setDestination(handleO(j,bi),(PE,x)); \\
& \qquad\qquad\qquad\qquad o_j \leftarrow (ownPE(),?) \\
&] \\
& \textbf{end for} \\
& \textbf{for } l \in \{1..n\} \textbf{ do} \qquad\qquad -- \text{ child's inports} \\
& (PE,f,x) \leftarrow o'_l; \\
& [\, realInport(j,bi) \quad \rightarrow i_l \leftarrow (ownPE(), createInport(createQM())); \\
& \qquad\qquad\qquad\qquad [outport(x) \rightarrow connectInport(i_l,o_l)]; \\
& \qquad\qquad\qquad\qquad [\neg f \rightarrow sendFinalDestination((PE,x),i_l)] \\
& \Box\, annotatedID(l,bi) \rightarrow (PE,f,x) \leftarrow o'_l; setOrigin(handleI(l,bi),(PE,x)); \\
& \qquad\qquad\qquad\qquad i_l \leftarrow (ownPE(),?) \\
&] \\
& \textbf{end for} \\
& tso \leftarrow getOwnIdentity() : \\
& saveDestinations(tso, i'_1, \ldots, i'_k); \\
& saveOutports(tso, o_1, \ldots, o_k); \\
& sendAck(PE_{parent}, o_{tso}, i'_1, \ldots, i'_k, o'_1, \ldots, o'_n, o'_1, i_1, \ldots, i_n, o_1, \ldots, o_k); \\
& a \leftarrow createTuple(b_1, \ldots, b_n); \\
& pushStack(a); \\
& enter(v)
\end{array}
$$

3.5 Reception of ACK message at parent's side

When the acknowledge message

$$ACK(PE_{child}, o_{tso}, i'_1, \ldots, i'_k, o'_1, \ldots, o'_n, i_1, \ldots, i_n, o_1, \ldots, o_k)$$

arrives to the parent's PE in a bypassing situation:

- Some of the father's outports may be annotated as being handles. We save them anyway so that the function `create?Thread` will detect it and will not create the corresponding thread.
- Some of the father inports may be annotated as being handles. In this case we should not do the *connectInport* done in the normal case.

- When we save the destinations, i_j, some of these may be of the form $(PE, ?)$ meaning that they correspond to handles in the child, so that when we go inside the code for `create?Thread` the thread will be created but not inmediately scheduled.

- If a parent's input/output is annotated as being a handle and the corresponding child's origin/destination is an actual outport/inport, we register this circumstance in the entry of the DBT associated to the handle. If this message is the second one related to this entry, that means that we have completed the actual origin and destination of the handle. So we send a BYPASS message to the receiver process.

In consequence, the code executed by the RTS is:

$tso \leftarrow getThread(o_{tso})$;
$saveDestinations(tso, i_1, \ldots, i_n)$;
$saveOutports(tso, o'_1, \ldots, o'_n)$;
for $j \in \{1..k\}$ **do** −− father's inports
  $(PE_p, f, x) \leftarrow i'_j$;
  $(PE_{ch}, y) \leftarrow o_j$;
  [$inport(x) \wedge outport(y) \rightarrow connectInport(x, o_j)$
  [] $handleB(x) \wedge outport(y) \rightarrow setOrigin(x, (PE_{ch}, y))$;
    $(PE, z) \leftarrow getDestination(x); disposeHandle(x)$
    $[(PE, z) \neq ? \rightarrow sendBypass((PE, z), (PE_{ch}, y))]$
  ]
end for ;
for $l \in \{1..n\}$ **do** −− father's outports
  $(PE_p, x) \leftarrow o'_l$;
  $(PE_{ch}, y) \leftarrow i_l$;
  $[handleB(x) \wedge inport(y) \rightarrow setDestination(x, (PE_{ch}, y))$;
    $(PE, z) \leftarrow getOrigin(x); disposeHandle(x)$
    $[(PE, z) \neq ? \rightarrow sendBypass((PE_{ch}, y), (PE, z))]$
  ]
end for ;
$scheduleThread(tso)$

3.6 Thread creation

The code can be the same both in parent and child sides. First, the outport and the destination of the thread is retrieved. The only case analysis needed is the following:

- If the outport is an actual one, a thread must be created. If, additionally, the destination is an actual inport, the thread must be inmediately scheduled.

- Otherwise, nothing should be done.

$$
\begin{array}{ll}
create?Thread & (e,l) \stackrel{\text{def}}{=} \\
& (PE_d, i) \leftarrow getDestination(l); \\
& (PE_o, o) \leftarrow getOutport(l); \\
& [\ \ outport(o) \wedge inport(i) \rightarrow tso \leftarrow createThread(e,(PE_d,i)); \\
& \qquad\qquad\qquad\qquad\quad connectOutport(o,tso); \\
& \qquad\qquad\qquad\qquad\quad scheduleThread(tso)] \\
& \square\ outport(o) \wedge handle(i) \rightarrow tso \leftarrow createThread(e, undefD); \\
& \qquad\qquad\qquad\qquad\quad connectOutport(o,tso) \\
& \square\ \texttt{otherwise} \rightarrow \textbf{skip} \\
&]
\end{array}
$$

3.7 Reception of the FINAL-ORIGIN message

If a message

$$FINAL-ORIGIN((PE_{sender}, outport_{sender}), (PE_{grandparent}, x))$$

is received in a PE, the RTS does a case analysis on x. If x is a handle and this is the second message received for it, it sends the BYPASS message to the destination of the handle. If it is an actual inport, the normal protocol is simplified and only the ACKBYPASS message is sent to the origin of the channel. So, it executes the following actions:

$$
\begin{array}{l}
[handleb(x) \rightarrow setOrigin(x,(PE_{sender}, outport_{sender})); \\
\qquad (PE,z) \leftarrow getDestination(x); disposeHandle(x) \\
\qquad [(PE,z) \neq\ ? \rightarrow sendBypas((PE,z),(PE_{sender}, outport_{sender}))] \\
\square\ inport(x) \rightarrow connectInport(x,(PE_{sender}, outport_{sender})); \\
\qquad sendAckBypass((PE_{sender}, outport_{sender}),(ownPE(),x)) \\
]
\end{array}
$$

3.8 Reception of the FINAL-DESTINATION message

If a message

$$FINAL-DESTINATION((PE_{receiver}, inport_{receiver}), (PE_{grandparent}, x))$$

is received in a PE, the RTS actions are symmetric to those of the FINAL-ORIGIN message. If x is an actual outport, the general protocol is simplified so that messages BYPASS and ACKBYPASS are saved. The thread associated to this outport is inmediately scheduled and it starts producing data messages. So, the following actions are executed:

$$
\begin{array}{l}
[handleb(x) \rightarrow setDestination(x,(PE_{receiver}, inport_{receiver})); \\
\qquad (PE,z) \leftarrow getOrigin(x); disposeHandle(x) \\
\qquad [(PE,z) \neq\ ? \rightarrow sendBypass((PE_{receiver}, inport_{receiver}),(PE,z))] \\
\square\ outport(x) \rightarrow tso \leftarrow getThread(x); \\
\qquad updateDestination(tso,(PE_{receiver}, inport_{receiver})); \\
\qquad scheduleThread(tso) \\
]
\end{array}
$$

3.9 Reception of the BYPASS message

If a message

$$BYPASS((PE_{receiver}, inport_{receiver}), (PE_{sender}, outport_{sender}))$$

arrives to a receiver PE, then the RTS connects the receiver inport to the sender outport and sends the ACKBYPASS message to the last one:

$PE_{receiver} \leftarrow ownPE()$;
$connectInport(inport_{receiver}, PE_{sender}, outport_{sender})$;
$sendAckBypass((PE_{sender}, outport_{sender}), (PE_{receiver}, inport_{receiver}))$;

3.10 Reception of the ACKBYPASS message

If an

$$ACKBYPASS((PE_{sender}, outport_{sender}), (PE_{receiver}, inport_{receiver}))$$

message arrives to a sender PE, then the RTS sets the destination in the thread that was waiting for this message and schedules it:

$tso \leftarrow getThread(outport_{sender})$;
$updateDestination(tso, (PE_{receiver}, inport_{receiver}))$;
$scheduleThread(tso)$