Encoding a Fragment of the π-Calculus in Haskell

Richard Fontana
Functional Programming

April 30, 2001

The pi-calculus of Milner, Parrow and Walker [1] has become a popular theoretical foundation for a broad class of concurrent computations. This process calculus is a notation for describing concurrrent computations as systems of communicating agents. Benjamin Pierce's programming language Pict [2, 3] is a programming language formed by adding high-level derived forms and a static type system to a syntactically-sugared pi-calculus core. Pict is, by design, intended to stand in the same relation to the pi-calculus as functional programming languages like Haskell do to the lambda-calculus.

I have tried to implement the roughest beginnings of a domain-specific language based on Haskell for an extremely tiny ("microscopic" might be more accurate) fragment of a Haskellized Pict, in the form of a very small library module containing several Haskell functions and datatype declarations. For convenience I will refer to the capabilities provided by this small module as "HPict".

The basic unit of computation in the pi-calculus is a *process*. The simplest process is called `Null` and has no observable behavior. The simplest complete HPict "program" is the following:

```
superrun [] Null
```

which produces no output. The function `superrun` is used to run complete programs. (The first argument can contain a concatenated series of typing restrictions for new variables, corresponding to the "new" construct in the pi-calculus. I have not yet implemented a type-checking feature in HPict.)

Two or more processes may be executed in parallel by separating them with the operator `:|:` :

```
superrun [] (Null :|: Null :|: Null)
```

In my implementation, an observable process is one which produces a Haskell screen output event. For example:

```
superrun [] (Event (putStr "abc"))
```

causes the string abc to be printed.

```
superrun []
       (Event (putStr "peering\n") :|: Event (putStr "absorbing\n") :|:
         Event (putStr "translating\n"))
```

```
translating
absorbing
peering
```

(This is of course symbolic concurrency only. The order of printing of arguments is deterministic.)

Besides processes, the other significant pi-calculus entities are channels (also called names), represented in HPict as strings. A channel is a port over which one process can send messages to another process. In HPict, the process expression `"x" :! (Tup [])` denotes a sender process that transmits a signal (represented as the empty value tuple) along the channel x. This transmission is completed when a receiver process is ready to accept a value along the same channel x. The receiver process has the form `"x" :? (Pat [], e)`, where `e` is a process expression that specifies what should be done after the signal has been received. When two such processes are placed in parallel, communication can take place:

```
superrun []
 (("x" :? (Pat [], Event (putStr "Got it!\n"))) :|: ("x" :! (Tup [])))
```

```
Got it!
```

In the expressions `"x" :! (Chan "y")` and `"x" :? (Var "z", e)`, the name `x` is, as in the previous signal example, the location where the two processes meet and exchange information. But in these expressions `(Chan "y")` is the argument of the message, and `(Var "z")` functions as the equivalent of a bound variable that, at the moment of communication, is replaced by the received value `(Chan "y")`.

When a receiving process obtains a channel, such as `Chan "y"`, by communication, it can use it for further communication. For example, a received channel can be subsequently used as a channel over which a value can be sent or received:

```
superrun []
 (("x":! (Chan "y")) :|:
```

```
        ("x" :? (Var "z", ("z" :! (Chan "u")))) :|:
        ("y" :? (Var "w", Event (putStr "Got it!\n"))))
```

```
Got it!
```

Or a received channel can be used subsequently as the value that is sent along some other channel:

```
      superrun []
        (("x" :! (Chan "y")) :|:
         ("x" :? (Var "z", ("a" :! (Chan "z")))) :|:
         ("a" :? (Var "w", Event (putStr "Got it!\n"))))
```

```
Got it!
```

In general, each communication step involves the atomic transmission of a single value from sender to receiver (entailing a matching algorithm in the process-running functions in the module). Our Value datatype is as follows:

```
data Value =   Chan Channel
             | Tup [Value]
             | String String  -- special value for output, not used here
             deriving (Eq, Show)
```

This represents that a value can be either a channel or a tuple of values. If "x" and "y" are Channels, then the following are Values:

```
Chan "x"
Tup [(Chan "x"), (Chan "y")]
Tup [(Chan "x"), (Tup [(Tup [(Chan "y"), (Chan "x")])]), (Chan "y"), (Tup
 []), (Chan "x")]
```

The `Process` datatype is represented as follows:

```
data Process = Null
             | Event (IO ())
             | Channel :! Value -- Sender process (send signal along channel);
             | Channel :? (Pattern, Process) -- Receiver process
             | Process :|: Process
             deriving Show
```

The general form of a sender process is `"x" :! v`, where `v` is a `Value`. Symmetrically, a receiver process has the form `"x" :? (p, e)`, where `p` is a `Pattern`:

```
data Pattern = Var Variable
             | Pat [Pattern]
             deriving (Eq, Show)
```

(where Variables are, like Channels, just aliases for Strings).

In principle, when a sender `x :! v` communicates with a receiver `x :? (p, e)`, the Value `v` is supposed to be matched against the Pattern `p` to yield a set of bindings for the variables in `p`. I have in fact written a function `bind` which yields such a set, following certain rules: any value v matches a variable pattern x (a singleton binding), and if p and v are composite, Pat [p1, ..., pn] and Tup [v1, ..., vn], and if for each i, the value vi matches the subpattern pi, we get a composite binding.

However, currently I do not use `bind` at the stage in which two potentially communicating processes are matched, though this is what should be done. I do use `bind` as an auxiliary function in the function `inst` which instantiates the bound variables of a receiver process expression. If `bind` were used at the initial process-matching stage it could function as an extremely primitive type-checker, catching processes that should not match because they have different "arities" (if that is the correct term here) of values and patterns at the highest composite level.

Type information is currently superfluous, though some appropriate type-manipulating functions have been written. Type-checking capabilities could be added to the module with a small amount of additional work. The relevant datatype is:

```
data Type = Uncertain
          | Comp [Type]
          | Carries Type
          deriving Show

signal = Comp []
```

It is contemplated that each value v has a type T (which at a particular point in processing might be `Uncertain` or unresolved). Tuple values have composite types. The idea is that a channel should be used to send and receive values of exactly one type. A channel has a `Carries` type (i.e., the type of the value it is permitted to carry).

A `new` function allows the typing of a new channel to be declared at the outset and kept in a list of channels and types that is supposed to be updated as a process is processed. I have not completed the task of including the type-updating/type-checking features in the functions that run a process; right now a list of channels and types can be provided at the outset but nothing is done with it.

Here is a final example which, while simple, gives some sense of the potential of expressiveness of such a language. The process

```
("b": ? (Pat [(Var "t"), (Var "f")], ("t":!(Tup[]))))
```

reads two channels from the channel "b" and transmits a signal along the first one. This can be seen as an encoding of the boolean value True, since it gives an outside observer the ability to select between two courses of action by interrogating this process along "b" and choosing the process's next action based on whether a response arrives along "t" or "f". Analogously, the value False is encoded as

```
("b": ? (Pat [(Var "t"), (Var "f")], ("f":!(Tup[]))))
```

If another process wants to test whether it is running in parallel with the first or the second of these processes, it transmits a pair of channels and then waits to see where a response appears:

```
superrun
   ( (new "b" (Carries (Comp [(Carries signal),(Carries signal)])) []) ++
     (new "t" (Carries signal) []) ++ (new "f" (Carries signal) []) )
                (
                  -- Encodes the Boolean value "False"
                 ("b":?(Pat[(Var"t"),(Var"f")],("f":!(Tup[]))))  :|:
                (
                  -- The testing process
                     ("b":!(Tup[(Chan"t"),(Chan"f")]))   :|:
                     ("t":?(Pat[],(Event (putStr"It's true\n")))) :|:
                     ("f":?(Pat[],(Event (putStr"It's false\n")))) ) )

It's false
```

```
superrun
   ( (new "b" (Carries (Comp [(Carries signal),(Carries signal)])) []) ++
     (new "t" (Carries signal) []) ++ (new "f" (Carries signal) []) )
                (
                  -- Encodes the Boolean value "True"
                 ("b":?(Pat[(Var"t"),(Var"f")],("t":!(Tup[]))))  :|:
                (
                  -- The testing process
                     ("b":!(Tup[(Chan"t"),(Chan"f")]))   :|:
                     ("t":?(Pat[],(Event (putStr"It's true\n")))) :|:
                     ("f":?(Pat[],(Event (putStr"It's false\n")))) ) )
```

```
It's true
```

In these examples I include the `new` channel/type declaration information even though it is ignored in the current state of the module.

References

[1] Milner, Robin, J. Parrow, and D. Walker. A calculus of mobile processes (Parts I and II). Information and Computation, 100:1-77, 1992.

[2] Pierce, Benjamin. Programming in the Pi-Calculus: A Tutorial Introduction to Pict (Pict Version 4.1). Computer Science Department, Indiana University, March 22, 1998. Available at ftp.cs.indiana.edu.

[3] Pierce, Benjamin, and David N. Turner. Pict Language Definition, Version 4.1. March 22, 1998. Available at ftp.cs.indiana.edu.