Seamless distributed computing

A short tale of an Erlang program

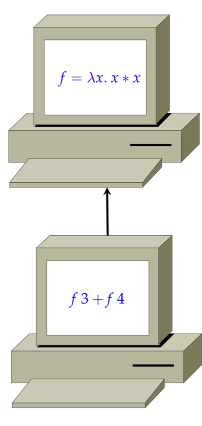
Let's say we have the following program:

$$\mathbf{let}\, f \;=\; \lambda x.\, x * x \;\mathbf{in}\, f\,3 + f\,4$$

Now we want the *f* function to run on a separate node in our distributed system. Let's write it in Erlang:

```
f(A_pid) ->
  receive X -> A_pid ! X * X end,
  f(A_pid).

main() ->
  F_pid = spawn(moduleName, f,
                [self()]),
  F_pid ! 3,
  receive X -> F_pid ! 4,
    receive Y -> X + Y end
  end.
```



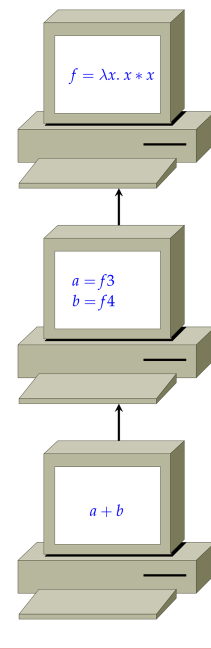
Configuration changes

On second thought, we actually need a proxy between the two nodes, so we rewrite the program:

```
c() ->
  receive {Pid, X} -> Pid ! X * X end,
  c().

b(A_pid, C_pid) ->
  receive
    requesta ->
      C_pid ! {self(), 3},
      receive X -> A_pid ! X end;
    requestb ->
      C_pid ! {self(), 4},
      receive X -> A_pid ! X end
  end,
  b(A_pid, C_pid).

main() ->
  C_pid = spawn(moduleName, c, []),
  B_pid = spawn(moduleName, b, [self(), C_pid]),
  B_pid ! requesta,
  receive X -> B_pid ! requestb,
    receive Y -> X + Y end
  end.
```



The problem

The *logic* of the program is the same, but we had to *rewrite* it for a simple configuration change.

The logic and communication details of the program are all mixed up.

Proposal

The first program could simply be:

$$(\mathbf{let}\, f \;=\; (\lambda x.\, x * x) \,@\, B \;\mathbf{in}\, f\,3 + f\,4) \,@\, A$$

Where *@B* is a *locus specification,* meaning that that part of the compiled program should be on the node named *B.*

And the second:

$$(\mathbf{let}\, f \;=\; (\lambda x.\, x * x) @ C \;\mathbf{in}\, (f\,3) @ B + (f\,4) @ B) @ A$$

We are creating a programming language where the program logic is separated from the communication, which is handled automatically, making this possible. This language is a variant of PCF, and the locus specifiers can work on *any* sub-term.

The typing and operational semantics are not affected by these specifiers:

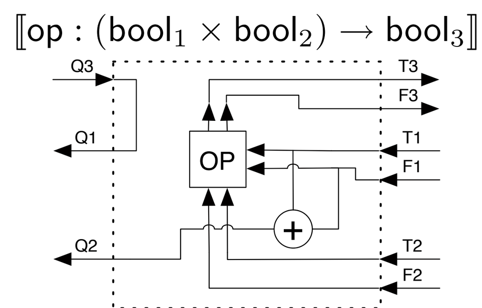
$$\frac{\Gamma \vdash t \,:\, \theta}{\Gamma \vdash t \,@\, A \,:\, \theta} \qquad\qquad \frac{t \Downarrow v}{t \,@\, A \Downarrow v}$$

Geometry of Interaction

Girard's *Geometry of Interaction* gives a syntax-free semantics for programs as token-passing networks of components"acting on the token.

Apart from being interesting from a semanticist's viewpoint, GoI has also inspired some interesting applications:

Dan Ghica et al.'s *Geometry of Interaction* series uses it to compile programs to hardware circuits.

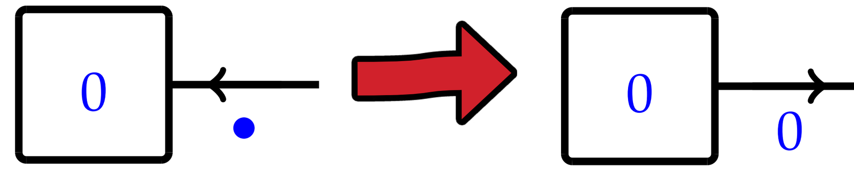


Ian Mackie's *Geometry of Interaction Machine* uses it for compiling programs to machine code.

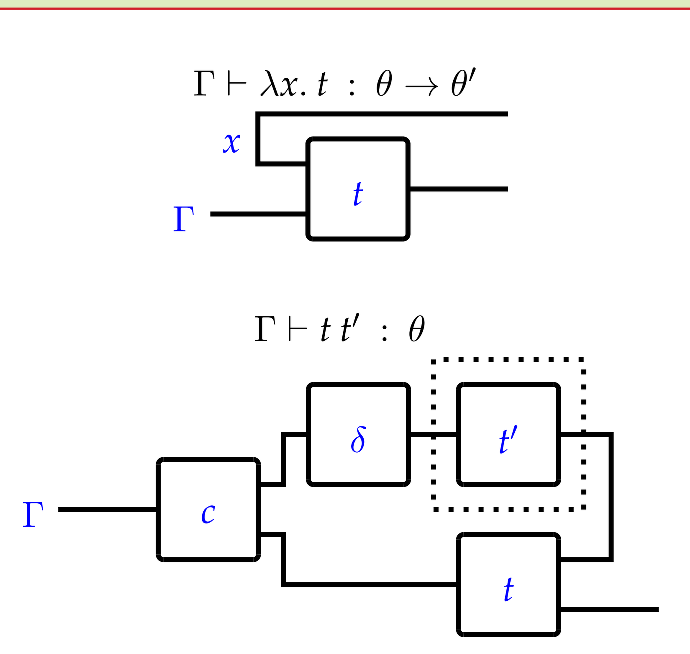
Here we are going to use it for compiling programs to networks of abstract machines.

Compilation

Components work in a simple request-answer manner, as illustrated by the component for the zero constant:

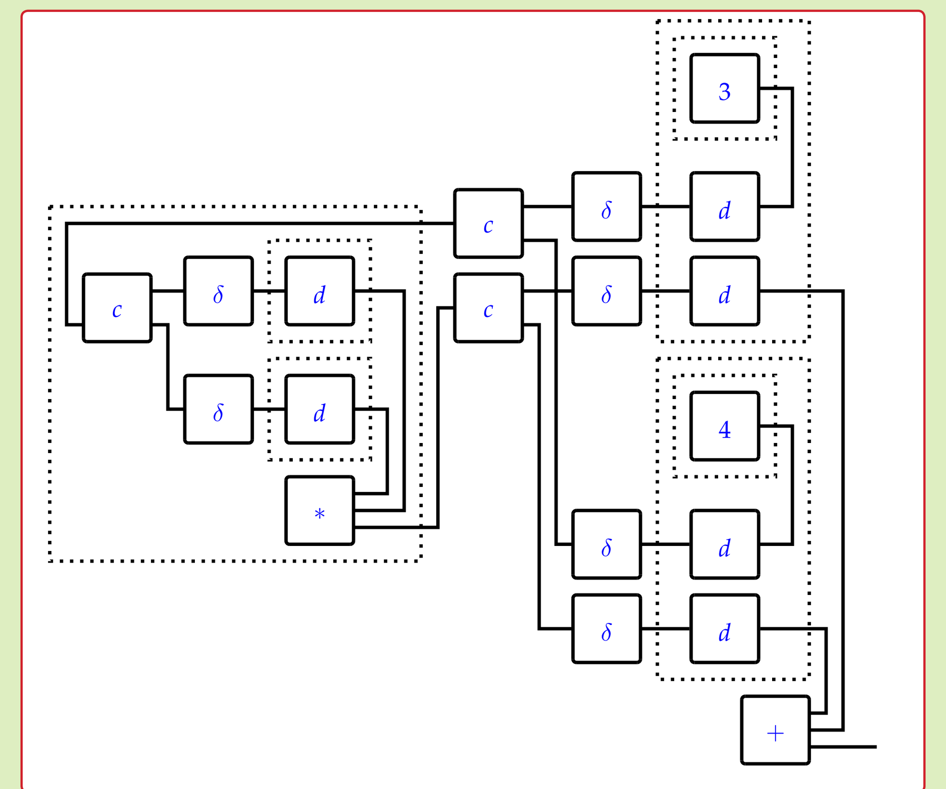


Terms of function type can request their argument, meaning that function application is done merely by communication:



Example

This is the result of compiling our initial program:

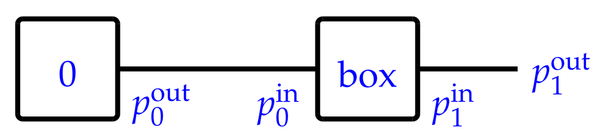


If we think of each box in the program as a node in the network, this will work. But it extremely fine-grained, and there is a lot of expensive communication even for this very simple program. What now?

Abstract machines

To make our components more computational and thus easy to compile, we construct abstract machines for describing their inner workings, and their interaction in a network.

The example below shows ports, labels and code:



$$0 = \langle \{p_0^{\text{out}} \mapsto l\}, \{l \mapsto \texttt{zero; send}\, p_0^{\text{in}}\} \rangle$$

$$\text{box} = \left\langle \begin{Bmatrix} p_1^{\text{in}} \mapsto l_1 \\ p_0^{\text{in}} \mapsto l_0 \end{Bmatrix}, \begin{Bmatrix} l_1 \mapsto \texttt{snd; send}\, p_0^{\text{out}} \\ l_0 \mapsto \texttt{unsnd; send}\, p_1^{\text{out}} \end{Bmatrix} \right\rangle$$

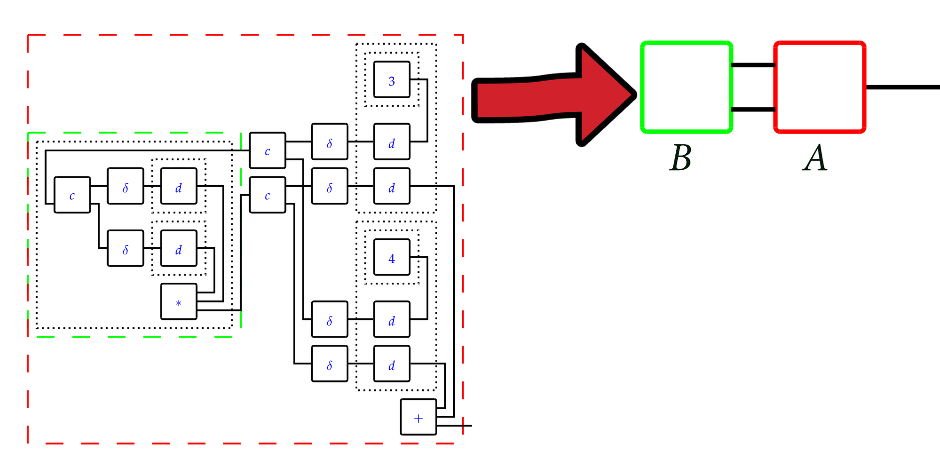
Combining components

We can arbitrarily combine abstract machines in a network gotten from compiling a term, obtaining networks of the granularity that we want.

$$combine(0, box) = \left\langle \{p_1^{\text{in}} \mapsto l_1\}, \begin{Bmatrix} l_1 \mapsto \texttt{snd; jump}\, l \\ l \mapsto \texttt{zero; jump}\, l_0 \\ l_0 \mapsto \texttt{unsnd; send}\, p_1^{\text{out}} \end{Bmatrix} \right\rangle$$

Going back to the example, we can combine components based on the locus specifiers:

$$(\mathbf{let}\, f \;=\; (\lambda x.\, x * x) \,@\, B \;\mathbf{in}\, f\,3 + f\,4) \,@\, A$$



Status

A prototype compiler has been created, which works by producing networks that run C programs communicating using MPI, following the abstract machine semantics.

The networks are first compiled into fine-grained networks which are combined according on the locus specifiers in the source code.

Future work

We are working on a separate description language for specifying aspects related to the communication, such as configuration, error-handling and code location. These things could then be changed without having to rewrite the logic of the program.

We are also adding constructs such as local variables and parallelism (essentially making the language Idealised Algol) to our language.

Olle Fredriksson
ohf162@cs.bham.ac.uk
The University of Birmingham