Seamless distributed computing from the Geometry of Interaction

Olle Fredriksson and Dan R. Ghica

The University of Birmingham, U.K.

Abstract. In this paper we present a seamless approach to writing and compiling distributed code. By "seamless" we mean that the syntax and semantics of the distributed program remain the same as if it was executed on one node only, except for label annotations indicating on what node sub-terms of the program are to be executed. There are no restrictions on how node labels are to be assigned to sub-terms. We show how the paradigmatic (higher-order functional recursive) programming language PCF, extended with node annotations, can be used for this purpose. The compilation technique is directly inspired by game semantics and the Geometry of Interaction.

1 Introduction

The conventional view of a program running in a distributed system, commonly called a *distributed program*, is of processes running on the nodes of a network, exchanging information by passing messages. This is the view supported by the ubiquitous *Message Passing Interface* system [15]. Although an effective method for writing distributed programs, MPI-style programming is far more laborious than programming a stand-alone computer. In order to create a distributed version of a program developed to run on a single node the code needs to be partitioned into modules that will run on each node in the distributed system, and the communication between the modules needs to be reduced to data communication and handled explicitly by the programmer.

A more abstract, and therefore more convenient, way of programming the interaction between process running on different nodes is through *Remote Procedure Call* [16]. In RPC, *communication* is subsumed by *function call*, which entails a certain protocol between the caller and the callee, i.e. first the arguments are sent by the caller along with the function name, then the result is returned by the callee. The success of RPC is reflected in the large number of variations it spawned, such as Java's Remote Method Invocation, SOAP, CORBA's object request broker, etc. With the advent of cloud computing more comprehensive frameworks were developed based on this idea, such as Microsoft's Windows Communication Framework, Google's Web Toolkit or Facebook's Thrift framework, and aimed specifically at web-delivered computational services. Although significantly more convenient that MPI-style programming, RPC-style programming still requires a certain amount of boilerplate code and, more importantly, it is not compatible with higher-order functions and functional programming.

A higher-order version of RPC seems to require the sending of functions (code) from the caller to the callee and vice versa, because functions can now be both arguments and results. One solution is that all nodes have access to a local instance of each function in the program and can send references to such functions (possibly paired with some free variables, forming a closure). Erlang [1] and Cloud Haskell [4] take this approach. Erlang, which runs in a virtual machine, even allows the sending of syntax trees for terms that do not exist on the remote node. However, both these approaches have the disadvantage that a program running on a single node needs to be "ported" to the distributed setting by including significant amounts of non-trivial boilerplate code.

What we will describe in this paper is a *seamless* approach to distributed programming: the distributed program is syntactically and semantically identical to the same program running on the single node, except for annotations (labels) indicating the names of the nodes where particular terms are to be executed. There is no language-induced restriction regarding the way locations are assigned: any syntactic sub-term of the program can be given an arbitrary node label, which will mean that it will be executed on the node of that label. There is no explicit communication between nodes, all the interaction being automatically handled "under the hood" by the generated code.

Example To illustrate this point consider the same program written in Erlang versus PCF annotated with location information. Consider the PCF program $\textbf{let}\, f \;=\; \lambda x.\, x * x \;\textbf{in}\; f\,3 + f\,4$, and suppose that we want to delegate the execution of the multiplication operation on a node C while the rest of the program executes on the (main) node A. The annotated PCF code is simply: $(\textbf{let}\, f \;=\; (\lambda x.\, x * x) \,@\, C \;\textbf{in}\; f\,3 + f\,4) \,@\, A$

In Erlang, things are much more complicated. The f function can be set up as a server which receives request messages:

```
c(A_pid) -> receive X -> A_pid ! X * X end, c(A_pid).
main() ->
  C_pid = spawn(f, c, [self()]), C_pid ! 3,
  receive X -> C_pid ! 4, receive Y -> X + Y end
  end.
```

Arguably, the logical structure of the program is lost in the detail. Moreover, if we want to further delegate the addition to a server B, the annotated PCF is $(\textbf{let}\, f \;=\; (\lambda x.\, x * x) \,@\, C \;\textbf{in}\; (f\,3 + f\,4) \,@\, B) \,@\, A$ whereas the Erlang version of the three-server distribution is even more complicated than the two-server version, which further obscures its logical structure:

```
c() -> receive {Pid, X} -> Pid ! X * X end, c().
b(A_pid, C_pid) ->
  receive
    request0 -> C_pid ! {self(), 3}, receive X -> A_pid ! X end;
    request1 -> C_pid ! {self(), 4}, receive X -> A_pid ! X end
  end,
  b(A_pid, C_pid).
main() ->
  C_pid = spawn(f2, c, []),
  B_pid = spawn(f2, b, [self(), C_pid]),
  B_pid ! request0,
  receive X -> B_pid ! request1, receive Y -> X + Y end
  end.
```

Contribution The main technical challenge we address in this paper is handling higher-order and recursive computation. To be able to give a focussed and technically thorough treatment we will handle the paradigmatic functional programming language PCF [13], a language which is well understood semantically and which lies at the foundation of practical programming languages such as Haskell.

Conceptually, what makes the seamless distribution of PCF programs possible is an interpretation inspired by game semantics [6] and the Geometry of Interaction (GoI) [10]. These models are built on the principle of reducing function call to communication and computation to interaction.

Note that the idea of reducing computation to interaction also appeared in the context of process calculi [11]. This was a development independent of game semantics and GoI which happened around the same time. Compilation of distributed languages via process calculi is also possible, in languages such as Pict [12], but the methodology is different. Whereas we aim to hide all the communication by making it implicit in function call, Pict and related languages embedded process calculus syntax to develop communication protocols. A further significant development in compilation based on process calculi was the idea of *mobile code*, where software agents are explicitly sent across the network between running processes [3, 14]. By contrast, in our methodology no code needs to be transmitted. Also note that GoI itself has been used before to compile PCF-like programs to unconventional architectures, namely reconfigurable digital circuits [7].[1]

The GoI model reduces a program to a static network of elementary nodes. Although this is eminently suitable for hardware synthesis, where the elementary nodes become elementary circuits and the network becomes the circuit interconnect, it is too fine grained for distributed compilation. We address this technical challenge by introducing a new style of *abstract machine* which has elementary instructions for both control (jumps) and communication. These machines can be (almost) arbitrarily *combined* by replacing communication with jumps, which gives a high degree of control over the granularity of the network.

Our compiler works in several stages. First it creates the fine-grained network of elementary communicating abstract machines. Then, using node annotations (labels), it combines all machines which arise out of the compilation of terms using the same label. The final step is to compile the abstract machines down to executable code using C for local execution and MPI for inter-machine communication.

[1] Tool available at http://veritygos.org.

2 PCF and its GoI model

We use conventional PCF with natural numbers as the only base type, extended with an annotation $t\ @\ A$, for specifying the locus of a term's computation. This annotation is to be thought of as a compiler directive: the operational semantics and typing rules simply ignore the locus specifier and are otherwise standard [13].

Girard's Geometry of Interaction [8] is a model for linear logic used for the study of the dynamics of computation, seeing a proof in the logic as a net, executed through the passing of a token along its edges. It is well known that GoI can be extended to also interpret programming languages and that it is useful in compiling programs to low-level machine code [10]. In this paper, we will use it as an interpretation for terms in our language, but we will use the notion of a proof net quite literally in that our programs will be compiled into networks of communicating nodes that operate on and send a token to each other.

We give an interpretation of terms in our language similar to the GoI interpretations by Hoshino [9] and Mackie [10], where terms are coded into linear logic proof nets. The only difference is that the interfaces of our nets are determined by type instead of being homogeneous. In this regard, the interpretation is more similar to the hardware circuits presented by the second author [5]. The reason why this works is because our language is not polymorphic, which otherwise necessitates that the interfaces are more homogeneous, since polymorphic terms may be instantiated at different types.

Term interpretations are built by connecting the ports of graphical components, that we think of as the nodes in our network. Two connected components can communicate bidirectionally through data tokens, defined by the grammar:

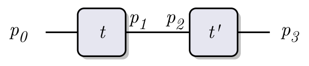
$$e ::= \bullet \mid 0 \mid \mathbf{S}\,e \mid \mathbf{inl}\,e \mid \mathbf{inr}\,e \mid \langle e, e\rangle.$$

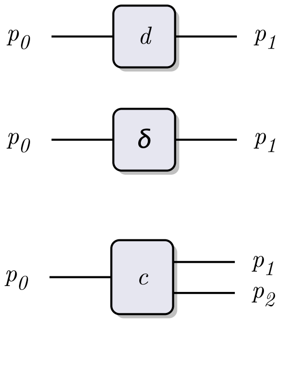
We first give a reading of these components as partial maps between data tokens. In the next section we will give a low-level description of their inner workings as communicating abstract machines.

The standard GoI components are given in Figure 1: dereliction (d), promotion (δ) and contraction (c). The components are bidirectional and their behaviour is given by a function mapping the values of a port at a given moment to their values at the next moment. We denote the value on a port which sends/receives no data as $\bot$. Two well-formedness conditions of GoI nets are that at most one port is not $\bot$ (i.e. a single-token is received at any moment) and $\overline{\bot} = (\bot, \ldots, \bot)$ is a fixed-point for any net (i.e. no spontaneous output is created).

Let π_2, π_2 be the first and second projections. Components are connected by functional composition (in both directions) on the shared port, represented graphically as:

$$(t; t')(e, \bot) = t'(\pi_2 \circ t(e, \bot), \bot)$$
$$(t; t')(\bot, e) = t(\bot, \pi_1 \circ t'(\bot, e)).$$





$$d(\langle \bullet, e\rangle, \perp) = (\perp, e)$$
$$d(\perp, e) = (\langle \bullet, e\rangle, \perp)$$

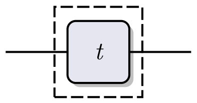
$$\delta(\langle\langle e, e'\rangle, e''\rangle, \perp) = (\perp, \langle e, \langle e', e''\rangle\rangle)$$
$$\delta(\perp, \langle e, \langle e', e''\rangle\rangle) = (\langle\langle e, e'\rangle, e''\rangle, \perp)$$

$$c(\langle \mathbf{inl}\, e, e'\rangle, \perp, \perp) = (\perp, \langle e, e'\rangle, \perp)$$
$$c(\langle \mathbf{inr}\, e, e'\rangle, \perp, \perp) = (\perp, \perp, \langle e, e'\rangle)$$
$$c(\perp, \langle e, e'\rangle, \perp) = (\langle \mathbf{inl}\, e, e'\rangle, \perp, \perp)$$
$$c(\perp, \perp, \langle e, e'\rangle) = (\langle \mathbf{inr}\, e, e'\rangle, \perp, \perp)$$

Fig. 1. Dereliction, promotion, and contraction

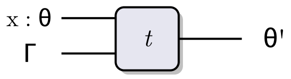
Exponentials Tokens need to carry both data and 'routing' information, but we want the basic components to have no access to the routing information but to act on data only. The role of the exponential functor (!) is to remove this routing information from the enclosed component, pass the data to the component, then restore the routing information. Diagrammatically this is represented as a dotted box around a network, defined formally below:

$$!t(\langle e, e'\rangle, \perp) = (\perp, \langle e, (\pi_2 \circ t(e', \perp))\rangle)$$
$$!t(\perp, \langle e, e'\rangle) = (\langle e, (\pi_1 \circ t(\perp, e))\rangle, \perp)$$



Types as interfaces The interface of a net is determined by the typing judgement of the term it interprets. The **nat** type corresponds to one port; the function type, $\theta \to \theta'$, induces an interface which is the disjoint union of those for θ, θ'. A typing environment $\Gamma = x_1 : \theta_1, \ldots x_n : \theta_n$ induces an interface which is the disjoint union of the interfaces for each θ_i. A term with typing judgement $\Gamma, x : \theta \vdash t : \theta'$ interface given by the environment on the left and its type on the right. Diagrammatically this is:

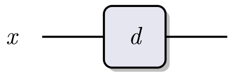
$$[\![x : \theta, \Gamma \vdash t : \theta']\!] =$$



Note that type $\theta \times \theta'$ can be interpreted, as convenient, either as a pair of ports or as a single port sending or receiving pair-tokens. Because of the well-formedness conditions the pairs are always of shape $(e, \perp)$ or $(\perp, e)$.

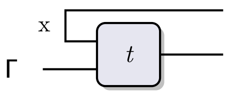
Terms as networks As the variables in the context correspond to the linear ! type, the GoI interpretation requires the use of dereliction:

$$[\![x : \theta, \Gamma \vdash x : \theta]\!] =$$



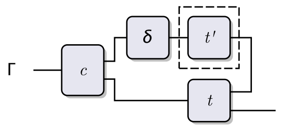
Abstraction and application When interpreting an abstraction, the variable is added to the context of the inner term, just like in the typing rule, and exposed in the interface of the final component. We only show the diagrammatic definition, the formalisation is obvious:

$$[\![\Gamma \vdash \lambda x.\, t \ : \ \theta \to \theta']\!] =$$



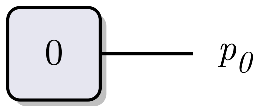
In the interpretation of application note the use of dereliction and exponentiation in the way the argument t' is connected to function t; this corresponds to the standard linear decomposition of call-by-name evaluation. Also note the use of contraction to explicitly 'share' the use of the free identifiers in Γ.

$$[\![\Gamma \vdash t\, t' \ : \ \theta']\!] =$$



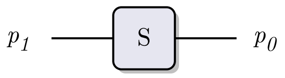
Constants The interpretation of a constant is a simple component that answers with 0 when requested, i.e. $\mathbf{0}(\bullet) = 0$. Diagrammatically,

$$[\![\Gamma \vdash 0 \ : \ \mathbf{nat}]\!] =$$



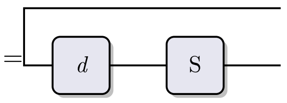
Successor A new component is needed for the interpretation of the successor function. This handles the **S** operation directly on a natural number.

$$\mathbf{S}\,(\bot, \bullet) = (\bullet, \bot)$$
$$\mathbf{S}\,(n, \bot) = (\bot, \mathbf{S}\, n).$$



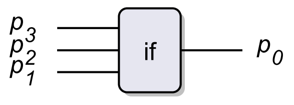
To make it possible to use this component in our interpretation it needs to be wrapped up in an abstraction and a dereliction to bring the argument to the right linear type.

$$[\![\Gamma \vdash \mathbf{S} \ : \ \mathbf{nat} \to \mathbf{nat}]\!] =$$



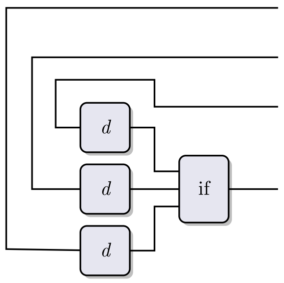
Conditionals Similar to how addition was handled, conditionals are done by constructing a new component and then wrapping it up as a function.

$$\begin{aligned}
\mathrm{if}(\bullet, \bot, \bot, \bot) &= (\bot, \bullet, \bot, \bot)\\
\mathrm{if}(\bot, 0, \bot, \bot) &= (\bot, \bot, \bullet, \bot)\\
\mathrm{if}(\bot, \mathbf{S}\, n, \bot, \bot) &= (\bot, \bot, \bot, \bullet)\\
\mathrm{if}(\bot, \bot, n, \bot) &= (n, \bot, \bot, \bot)\\
\mathrm{if}(\bot, \bot, \bot, n) &= (n, \bot, \bot, \bot).
\end{aligned}$$



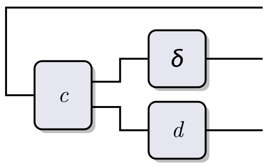
For the final interpretation of conditionals in the language, dereliction are added as the arguments are of exponential type:

$$[\![\Gamma \vdash \mathbf{if} \ \cdot \ \mathbf{then} \ \cdot \ \mathbf{else} \cdot \ : \mathbf{nat} \to \mathbf{nat} \to \mathbf{nat} \to \mathbf{nat}]\!] =$$



Recursion Recursion is interpreted as a component that is connected to itself as done by Mackie [10].

$$[\![\Gamma \vdash \mathbf{Y} \ : (\theta \to \theta) \to \theta]\!]$$



The abstract token machine interpretation given in this section is known to be sound [9, 10].

Theorem 1 (GoI Soundness). *Let $\vdash t : \mathbf{nat}$ be a closed PCF program (closed ground-type term) and $[\![t]\!]$ its GoI abstract-token machine representation. If t evaluates to n ($t \Downarrow n$) then $[\![t]\!](\bullet) = n$.*

3 The SIC machine

To be able to describe the inner workings of the components and see how they can be compiled to executables we construct an abstract machine, the Stack-Interaction-Control (SIC) machine, which has a small instruction set tailor made for that purpose. The SIC machine works similar to Mackie's [10] but with the important distinction that it also allows sending and receiving messages to and from other machines, to model networked distribution. The machine descriptions and configurations are specified in Figure 2. An initial configuration for a machine description $\langle P, L\rangle$ is given by the function *initial*$(\langle P, L\rangle) = \langle \mathbf{passive}, \bullet, P, L\rangle$.

3.1 SIC semantics

The semantics of the machines are given as a transition relation in Figure 3. The machine instructions make it possible to manipulate the data token of an active machine, using the stack for state and intermediate results. The last two rules handle sending and receiving messages.

Label	l		
Port	p		
Instruction	$I ::=$	`inl` | `inr`	Tags
		| `fst` | `snd`	Projections
		| `unfst` | `unsnd`	Reverse projections
		| `flip` | `push` | `pop`	Stack operations
		| `zero` | `suc`	Operations on natural numbers
Code	$C ::=$	$I; C$	Instruction cons
		| `jump` l	Jump to label l
		| `match` $l_1\ l_2$	Conditional jump
		| `if` $l_1\ l_2$	Conditional jump (nat)
		| `send` p	Send on port p
Stack	$S ::=$	$\bullet$	Empty
		| $d :: S$	Cons
Port map	$P : p \to l$		
Label map	$L : l \to C$		
Machine description	$::=$	$\langle P, L\rangle$	
Machine state	$M_{\text{state}} ::=$	**active** $C\ d$	
		| **passive**	
Machine configuration	$M ::=$	$\langle M_{\text{state}},\ S,\ P,\ L\rangle$	

Fig. 2. SIC specification

Now it is possible to define a machine network where many of these machines can operate together. This machine network is described in the style of the Chemical Abstract Machine [2]. A machine network is simply a multiset ($\mathcal{M}$) of messages (port-data pairs (p, d)) that are "in the air" and a list of machine configurations (M). The transition relation for the network is:

$$\frac{M \to M'}{\langle \mathcal{M} \mid N_1, M, N_2\rangle \to \langle \mathcal{M} \mid N_1, M', N_2\rangle}\ \textsc{Silent}$$

$$\frac{M \xrightarrow{\texttt{send}\ \langle p,d\rangle} M'}{\langle \mathcal{M} \mid N_1, M, N_2\rangle \to \langle \mathcal{M} \uplus \{\langle p, d\rangle\} \mid N_1, M', N_2\rangle}\ \textsc{Send}$$

$$\frac{M \xrightarrow{\overline{\texttt{send}\ \langle p,d\rangle}} M'}{\langle \mathcal{M} \uplus \{\langle p, d\rangle\} \mid N_1, M, N_2\rangle \to \langle \mathcal{M} \mid N_1, M', N_2\rangle}\ \textsc{Receive}$$

Let *active*(N) be a function that returns all the machines in N that are in **active** state.

Proposition 1. *For any sets of machine configurations N and N' and multisets of messages $\mathcal{M}$and $\mathcal{M}'$, if $\langle \mathcal{M} \mid N\rangle \to \langle \mathcal{M}' \mid N'\rangle$, then $|\mathcal{M}'| + |\text{active}(N')| = |\mathcal{M}| + |\text{active}(N)|$.*

Proof. By cases on the machine network step relation and construction of the SIC machine's step relation.

$$
\begin{array}{lcl}
\langle \textbf{active}\,(\texttt{inl};\,C)\; d,\; S,\; P,\; L\rangle & \to & \langle \textbf{active}\, C\; (\textbf{inl}\, d),\; S,\; P,\; L\rangle \\
\langle \textbf{active}\,(\texttt{inr};\,C)\; d,\; S,\; P,\; L\rangle & \to & \langle \textbf{active}\, C\; (\textbf{inr}\, d),\; S,\; P,\; L\rangle \\
\langle \textbf{active}\,(\texttt{fst};\,C)\; \langle d_1, d_2\rangle,\; S,\; P,\; L\rangle & \to & \langle \textbf{active}\, C\; d_1,\; d_2 :: S,\; P,\; L\rangle \\
\langle \textbf{active}\,(\texttt{snd};\,C)\; \langle d_1, d_2\rangle,\; S,\; P,\; L\rangle & \to & \langle \textbf{active}\, C\; d_2,\; d_1 :: S,\; P,\; L\rangle \\
\langle \textbf{active}\,(\texttt{unfst};\,C)\; d_1,\; d_2 :: S,\; P,\; L\rangle & \to & \langle \textbf{active}\, C\; \langle d_1, d_2\rangle,\; S,\; P,\; L\rangle \\
\langle \textbf{active}\,(\texttt{unsnd};\,C)\; d_2,\; d_1 :: S,\; P,\; L\rangle & \to & \langle \textbf{active}\, C\; \langle d_1, d_2\rangle,\; S,\; P,\; L\rangle \\
\langle \textbf{active}\,(\texttt{flip};\,C)\; d,\; d_1 :: d_2 :: S,\; P,\; L\rangle & \to & \langle \textbf{active}\, C\; d,\; d_2 :: d_1 :: S,\; P,\; L\rangle \\
\langle \textbf{active}\,(\texttt{push};\,C)\; d,\; S,\; P,\; L\rangle & \to & \langle \textbf{active}\, C\; \bullet,\; d :: S,\; P,\; L\rangle \\
\langle \textbf{active}\,(\texttt{pop};\,C)\; d_1,\; d_2 :: S,\; P,\; L\rangle & \to & \langle \textbf{active}\, C\; d_1,\; S,\; P,\; L\rangle \\
\langle \textbf{active}\,(\texttt{zero};\,C)\; d,\; S,\; P,\; L\rangle & \to & \langle \textbf{active}\, C\; 0,\; S,\; P,\; L\rangle \\
\langle \textbf{active}\,(\texttt{suc};\,C)\; n,\; S,\; P,\; L\rangle & \to & \langle \textbf{active}\, C\; \mathbf{S}\, n,\; S,\; P,\; L\rangle \\
\langle \textbf{active}\,(\texttt{jump}\, l)\; d,\; S,\; P,\; L\rangle & \to & \langle \textbf{active}\, L(l)\; d,\; S,\; P,\; L\rangle \\
\langle \textbf{active}\,(\texttt{match}\, l_1\, l_2)\; (\textbf{inl}\, d),\; S,\; P,\; L\rangle & \to & \langle \textbf{active}\, L(l_1)\; d,\; S,\; P,\; L\rangle \\
\langle \textbf{active}\,(\texttt{match}\, l_1\, l_2)\; (\textbf{inr}\, d),\; S,\; P,\; L\rangle & \to & \langle \textbf{active}\, L(l_2)\; d,\; S,\; P,\; L\rangle \\
\langle \textbf{active}\,(\texttt{if}\, l_1\, l_2)\; 0,\; S,\; P,\; L\rangle & \to & \langle \textbf{active}\, L(l_2)\; \bullet,\; S,\; P,\; L\rangle \\
\langle \textbf{active}\,(\texttt{if}\, l_1\, l_2)\; \mathbf{S}\, n,\; S,\; P,\; L\rangle & \to & \langle \textbf{active}\, L(l_1)\; \bullet,\; S,\; P,\; L\rangle \\
\langle \textbf{active}\,(\texttt{send}\, p)\; d,\; S,\; P,\; L\rangle & \xrightarrow{\texttt{send}\;\langle p,d\rangle} & \langle \textbf{passive},\; S,\; P,\; L\rangle \\
\langle \textbf{passive},\; S,\; P,\; L\rangle & \xrightarrow{\overline{\texttt{send}\;\langle p,d\rangle},\; p\in dom(P)} & \langle \textbf{active}\, L(P(p))\; d,\; S,\; P,\; L\rangle
\end{array}
$$

Fig. 3. SIC step relation

Case Silent : In this rule, $\mathcal{M}' = \mathcal{M}$, so $|\mathcal{M}'| = |\mathcal{M}|$. The SIC machine M that takes a step goes from **active** to **active** since all SIC machine step rules that do not send or receive messages are on that form, which also implies that $|active(N)| = |active(N')|$.

Case Send : Here $\mathcal{M}' = \mathcal{M} \uplus \{\langle p, d\rangle\}$ so $|\mathcal{M}'| = |\mathcal{M}| + 1$. The only SIC machine rule that applies here is the send rule, which takes the machine M from state **active** to **passive** . So $|active(N')| = |active(N)| - 1$, and thus $|\mathcal{M}'| + |active(N')| = |\mathcal{M}| + 1 + |active(N)| - 1 = |\mathcal{M}| + |active(N)|$

Case Receive : In this case, $\mathcal{M} = \mathcal{M}' \uplus \{\langle p, d\rangle\}$ so $|\mathcal{M}'| = |\mathcal{M}| - 1$. The only SIC machine rule that applies is the receive rule, which takes the machine M from state **passive** to **active** , meaning that $|active(N')| = |active(N)| + 1$. Thus $|\mathcal{M}'| + |active(N')| = |\mathcal{M}| - 1 + |active(N)| + 1 = |\mathcal{M}| + |active(N)|$.

In particular, this proof means that if we start out with one message and no active machines, there can be at most one active machine at any point in the network's execution – the execution is *single-token.*

3.2 Components as SIC code

In each of the components we take all ports p_x and labels l_x to be distinct. Components are connected by giving an output port the same name as the input port of the component that it is connected to. To emphasise the input/output role of a port we sometimes write them as p_x^i when serving as input and p_x^o when serving as output. The machine descriptions for the different components are described by giving their port mappings P and label mappings L as a tuple.

The following three machines are stateless. They use the stack internally for intermediate results, but ultimately return the stack to the initial empty state.

Dereliction, d, removes the first component of the token tuple when going from left to right, and adds it when going in the other direction:

$$\text{dereliction} = \left\langle \begin{array}{l} p_0^i \mapsto l_0 \\ p_1^i \mapsto l_1 \end{array} \quad , \quad \begin{array}{l} l_0 \mapsto \texttt{snd};\ \texttt{pop};\ \texttt{send}\, p_1^o \\ l_1 \mapsto \texttt{push};\ \texttt{unfst};\ \texttt{send}\, p_0^o \end{array} \right\rangle$$

Promotion, δ, reassociates the data token to go from $\langle\langle e, e'\rangle, e''\rangle$ to $\langle e, \langle e', e''\rangle\rangle$ and back:

$$\text{promotion} = \left\langle \begin{array}{l} p_0^i \mapsto l_0 \\ p_1^i \mapsto l_1 \end{array} \quad , \quad \begin{array}{l} l_0 \mapsto \texttt{fst};\ \texttt{snd};\ \texttt{flip};\ \texttt{unfst};\ \texttt{unsnd};\ \texttt{send}\, p_1^o \\ l_1 \mapsto \texttt{snd};\ \texttt{fst};\ \texttt{flip};\ \texttt{unsnd};\ \texttt{unfst};\ \texttt{send}\, p_0^o \end{array} \right\rangle$$

Contraction, c, uses matching on the first component of the token to choose the right port to send on when going from left to right.

$$\text{contraction} = \left\langle \begin{array}{l} p_0^i \mapsto l_0 \\ p_1^i \mapsto l_1 \\ p_2^i \mapsto l_2 \end{array} \quad , \quad \begin{array}{l} l_0 \mapsto \texttt{fst};\ \texttt{match}\, l_3\, l_4 \\ l_1 \mapsto \texttt{fst};\ \texttt{inl};\ \texttt{unfst};\ \texttt{send}\, p_0^o \\ l_2 \mapsto \texttt{fst};\ \texttt{inr};\ \texttt{unfst};\ \texttt{send}\, p_0^o \\ l_3 \mapsto \texttt{unfst};\ \texttt{send}\, p_1^o \\ l_4 \mapsto \texttt{unfst};\ \texttt{send}\, p_2^o \end{array} \right\rangle$$

Dotted boxes with k inputs are defined as follows: Let $P_{\text{in}}^i = \{p_0^i, \ldots, p_{k-1}^i\}$, $P_{\text{out}}^i = \{p_k^i, \ldots, p_{2k-1}^i\}$, Let $P_{\text{in}}^o = \{p_0^o, \ldots, p_{k-1}^o\}$, $P_{\text{out}}^o = \{p_k^o, \ldots, p_{2k-1}^o\}$, and $L_{\text{out}} = \{l_k, \ldots, l_{2k-1}\}$. The box for these sets of ports and labels is then the following (note that boxes use the stack for storing their state):

$$\text{box} = \left\langle p_i^i \mapsto l_i \mid p_i^i \in P_{\text{in}}^i \cup P_{\text{out}}^i \quad , \quad \begin{array}{l} l_i \mapsto \texttt{snd};\ \texttt{send}\, p_{i+k}^o \mid l_i \in L_{\text{in}} \\ l_i \mapsto \texttt{unsnd};\ \texttt{send}\, p_{i-k}^o \mid l_i \in L_{\text{out}} \end{array} \right\rangle$$

For a constant, the component's abstract machine is defined as follows:

$$\text{constant} = \left\langle p_0^i \mapsto l_0 \quad , \quad l_0 \mapsto \texttt{zero};\ \texttt{send}\, p_0^o \right\rangle$$

The successor machine first asks for its argument, then runs the successor instruction on that.

$$\text{suc} = \left\langle \begin{array}{l} p_0^i \mapsto l_0 \\ p_1^i \mapsto l_1 \end{array} \quad , \quad \begin{array}{l} l_0 \mapsto \texttt{send}\, p_1^o \\ l_1 \mapsto \texttt{suc};\ \texttt{send}\, p_0^o \end{array} \right\rangle$$

The conditional uses the matching instruction `if` to choose the right branch depending on the natural number of the token.

$$\text{if} = \left\langle \begin{array}{l} p_0^i \mapsto l_0 \\ p_1^i \mapsto l_1 \\ p_2^i \mapsto l_2 \\ p_3^i \mapsto l_2 \end{array} \quad , \quad \begin{array}{l} l_0 \mapsto \texttt{send}\, p_1^o \\ l_1 \mapsto \texttt{if}\, l_{\text{true}}\, l_{\text{false}} \\ l_{\text{true}} \mapsto \texttt{send}\, p_2^o \\ l_{\text{false}} \mapsto \texttt{send}\, p_3^o \\ l_2 \mapsto \texttt{send}\, p_0^o \end{array} \right\rangle$$

To connect the port p_0 of machine M to p_1 of M' we rename them in the machine definitions to the same port name p. A port must be connected to at most one other port; in this case the net is said to be *deterministic*, as each message will be received by at most one other machine. A port of a machine which is not connected to a port of another machine is said to be a port *of the network*. By $\mathit{inputs}(M)$ ($\mathit{outputs}(M)$) we mean the inputs (outputs) of a machine, whereas by $\mathit{inputs}(N)$ ($\mathit{outputs}(N)$) we mean the inputs (outputs) of a net. Similarly, by $\pi(M)$ ($\pi(N)$) we mean the ports of a machine (network).

Theorem 2 (Soundness). *Let* $\vdash t : \mathbf{nat}$ *be a closed PCF program (closed ground-type term),* $[\![t]\!]$ *its GoI abstract-token machine representation and* N *its SIC-net implementation. If* t *evaluates to* n ($t \Downarrow n$) *then* $[\![t]\!](\bullet) = n$ *and* $\langle \{\langle p^i, \bullet\rangle\} \mid N\rangle \to^* \langle \{\langle p^o, n\rangle\} \mid N\rangle$.

4 Combining machines

When writing distributed applications, the location at which a computation is performed is vital. Traditional approaches are sometimes explicit about that, for instance by using message passing. Using our current term interpretation, and thinking of each abstract machine as running on a different node in a network, we get the communication in the network handled automatically, but will have one abstract machine for each (very small) component. The interpretation produces extremely fine grained networks where each node does very little work before passing the token along to another node. It is expected that the communication is one of the most performance critical parts in a distributed network, which is why it would be better if bigger chunks of computations happened on the same node before the token was passed along.

To make this possible, we devise a way to combine the descriptions of two abstract machines in a deterministic network to get a larger abstract machine with the same behaviour as the two original machines. Informally, the way to combine two machines is to remove ports that are used internally between the two machines (if any) and replace sends on those ports with jumps. The algorithm for combining components $M_1 = \langle P_1, L_1\rangle$ and $M_2 = \langle P_2, L_2\rangle$ is described formally below.

We use Δ for the symmetric difference of two sets. If $f : A \to B$ is a function we write as $f \restriction A'$ the restriction of f to the domain $A' \subseteq A$ and we extend it in the obvious way to relations. We use the standard notation $C[s/s']$ to denote the replacing of all occurrences of a string s by s' in C. We write $C[s(x)/s'(x) \mid x \in A]$ to denote the substitution of all strings of shape $s(x)$ by strings of shape $s'(x)$ with x in a list A, defined inductively as

$$C[s(x)/s'(x) \mid x \in \emptyset] = C$$
$$C[s(x)/s'(x) \mid x \in a :: A] = \big(C[s(a)/s'(a)]\big)[s(x)/s'(x) \mid x \in A]$$

The combination of two machines is defined by keeping the ports which are not shared and by replacing in the code the send operations to shared ports by jumps to labels given by the port mappings.

$$\begin{aligned}\mathit{combine}(M_1, M_2) = \langle &(P_1 \cup P_2) \restriction (\pi(M_1) \,\Delta\, \pi(M_2)),\\ &\qquad (L_1 \cup L_2)[\texttt{send}\, p/\texttt{jump}\, P(p) \mid p \in \pi(M_1) \cap \pi(M_2)]\rangle.\end{aligned}$$

There are two abuses of notation above. First, the union $P_1 \cup P_2$ above is on functions taken as sets of pairs and it may not result in a proper function. However, the restriction to $\pi(M_1) \,\Delta\, \pi(M_2)$ always produces a proper function. Second, $\pi(M_1) \cap \pi(M_2)$ is a set and not a list. However, the result of this substitution is independent of the order in which the elements of this set are taken from any of its possible list representations.

A network is said to be combinable if combining any of its components does not change the overall network behaviour

Definition 1. *A deterministic network of machines $N = M_1, \ldots, M_k$ is* combinable *if whenever*

$$\langle \{\langle p, d\rangle\} \mid N\rangle \to^* \langle \{\langle p', d'\rangle\} \mid N'\rangle$$

for some N', p in inputs(N)*, p' in* outputs(N)*, then for any $N_{combined}$ obtained from N by replacing M_i, M_j with* combine(M_i, M_j) *for some $i \neq j$ we have that*

$$\langle \{\langle p, d\rangle\} \mid N_{combined}\rangle \to^* \langle \{\langle p', d'\rangle\} \mid N'_{combined}\rangle$$

for some $N'_{combined}$.

Note that the combined net is not equivalent to the original net (for a suitable notion of equivalence such as bisimilarity) because it will have fewer observable messages being exchanged.

Lemma 1. *If a net N is combinable then $N_{combined}$ is also combinable.*

The set of combinable machines is hard to define exactly, so we would just like to find a sound characterisation of such machines which covers all the basic components we used and their combinations.

Definition 2. *A machine description $M = \langle P, L\rangle$ is* stack-neutral *if for all stacks S and S', p in* inputs(M)*, p' in* outputs(M)*, if*

$$\langle \{\langle p, d\rangle\} \mid \langle \mathbf{passive}, S, P, L\rangle\rangle \to^* \langle \{\langle p', d'\rangle\} \mid \langle \mathbf{passive}, S', P, L\rangle\rangle$$

then $S = S'$.

Definition 3. *A machine network N of k machines described by port mappings P_i and label mappings L_i is* stack-neutral*, if for all stacks S_i and S'_i, p in* inputs(N)*, p' in* outputs(N)*, if*

$$\begin{aligned}\langle \{\langle p, d\rangle\} \mid [\langle \mathbf{passive}, S_i, P_i, L_i\rangle \mid i \in \{1, \ldots, k\}]\rangle &\to^*\\ \langle \{\langle p', d'\rangle\} \mid [\langle \mathbf{passive}, S'_i, P_i, L_i\rangle \mid i \in \{1, \ldots, k\}]\rangle&\end{aligned}$$

then all $S_i = S'_i$.

Note that this definition is more general than having a list of stack-neutral machines, as a stack-neutral network's machines may use the stack for state after it has been exited as long as it's cleared before an output on a network port.

Proposition 2. *If two machine networks N_1 and N_2 (of initially passive machines) are stack-neutral, combinable and N_1, N_2 is deterministic, then N_1, N_2 is stack-neutral and combinable.*

For any SIC-net N let $box(N)$ be N with an additional box machine M with input ports $outputs(N)$ and output ports $inputs(N)$, defined as in Sec. 3.2.

Proposition 3. *If a machine network N is stack-neutral and combinable then* $\mathrm{box}(N)$ *is stack-neutral and combinable,*

From the following two results it follows by induction on the structure of the generated nets that

Theorem 3. *If $\Gamma \vdash t : \theta$ is a PCF term, $[\![t]\!]$ its GoI abstract-token machine representation and N the implementation of $[\![t]\!]$ as a SIC-net then N is combinable.*

With the ability to combine components, we can now exploit the t @ A annotations in the language. They make it possible to specify where a piece of code should be located (A is a node identifier). When this construct is encountered in compilation, the components generated in compiling t are tagged with A (possibly overwriting older tags).

Next, the components with the same tag are combined using the algorithm above and their combined machine placed on the node identified by the tag. This allows the programmer to arbitrarily choose where the compiled representation of a part of a term is placed. Soundness (Thm. 2) along with the freedom to combine nets (Thm. 3) ensures that the resulting network is a correct implementation of any (terminating) PCF program.

5 Compiling PCF

We developed an experimental compiler that compiles to C, using MPI for communication, using SIC abstract machines as an intermediate formalism. [2] Each machine description in a network is mapped to a C source file in a fairly straight-forward manner, using a function for each machine instruction and global variables for the data token and the stack. An example of a predefined instruction is that for the `flip` instruction:

```
inline void flip() {
  Data d1 = pop_stack();
  Data d2 = pop_stack();
  push_stack(d1);
  push_stack(d2);}
```

[2] Download from http://veritygos.org/dpcf

An abstract machine's label l corresponds to a C function `void l()` whose definition is a list of calls to the predefined machine instruction functions. In this representation, jumps are function calls. All functions are small and not used recursively so can be efficiently inlined.

Each process in MPI has a unique identifier called its *rank*, and messages can also be assigned a *tag*. A port in a SIC machine is uniquely determined by its tag, but also has to be assigned a rank so that the message can be sent to the correct node. This is resolved at compile-time. The main loop for a machine listening on ports corresponding to tags 0 and 1 looks like this:

```
while(1) {
  int port = receive();
  switch (port) {
    case 0: l0(); break;
    case 1: l1(); break;
    default: break;}}
```

Here `l0` and `l1` are functions corresponding to the labels associated with the ports. The predefined function `receive` calls `MPI_Recv`, which is an MPI function that blocks until a message is received. A process in this state thus corresponds to a machine in **passive** state. Upon receiving a message, the `receive` function de-serialises the message and assigns it to the global data token variable before returning the message's tag. The predefined function for the **send** instruction now has to take two parameters: the destination node's rank and the port's tag:

```
   inline void send(int node, int port);
```

The function takes care of serialising the data token sending it to the correct node using `MPI_Send`.

When all machines have been compiled to C, these can in turn be compiled to executables and run on different machines in a network where they use message passing for communication.

6 Conclusion and future work

We have shown a programming language and compilation model for seamless distributed computing, that provides freedom in choosing the location at which a computation takes place with implicitly handled communication. This was achieved by basing the model on the Geometry of Interaction and constructing a way to produce nodes that are more coarse grained than the standard elementary nodes, and showing that this is still correct.

So far semantics of a term are sequential – there is nothing taking place in parallel. The next step will be to investigate how to extend the system with a safe and flexible parallelisation mechanism. A start is to identify internally sequential partitions of the network that can safely be run in parallel. Then the evaluation of a function's arguments can be performed in parallel with the function as long as they are in different partitions of the network. Another idea is to add a more specific construct for parallelisation, e.g. one for map-reduce.

An important point that has not been discussed in this paper is fault-tolerance: A distributed system needs to be able withstand nodes crashing, becoming unavailable or being added by dynamically reassigning the locus of execution. This is also something that we would like to look into in the future.

References

1. J. Armstrong. A history of Erlang. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 6–1–6–26, New York, NY, USA, 2007. ACM.
2. G. Berry and G. Boudol. The chemical abstract machine. *Theor. Comput. Sci.*, 96(1):217–248, 1992.
3. A. Carzaniga, G. P. Picco, and G. Vigna. Designing distributed applications with mobile code paradigms. In W. R. Adrion, A. Fuggetta, R. N. Taylor, and A. I. Wasserman, editors, *ICSE*, pages 22–32. ACM, 1997.
4. J. Epstein, A. P. Black, and S. L. P. Jones. Towards Haskell in the cloud. In K. Claessen, editor, *Haskell*, pages 118–129. ACM, 2011.
5. D. R. Ghica. Geometry of Synthesis: a structured approach to VLSI design. In M. Hofmann and M. Felleisen, editors, *POPL*, pages 363–375. ACM, 2007.
6. D. R. Ghica. Applications of game semantics: From program analysis to hardware synthesis. In *LICS*, pages 17–26. IEEE Computer Society, 2009.
7. D. R. Ghica. Function interface models for hardware compilation. In *Formal Methods and Models for Codesign (MEMOCODE), 2011 9th IEEE/ACM International Conference on*, pages 131–142. IEEE, 2011.
8. J. Girard. Geometry of interaction 1: Interpretation of system F. *Studies in Logic and the Foundations of Mathematics*, 127:221–260, 1989.
9. N. Hoshino. A modified GoI interpretation for a linear functional programming language and its adequacy. In M. Hofmann, editor, *FOSSACS*, volume 6604 of *Lecture Notes in Computer Science*, pages 320–334. Springer, 2011.
10. I. Mackie. The geometry of interaction machine. In *POPL*, pages 198–208, 1995.
11. R. Milner. Functions as processes. *Automata, Languages and Programming*, pages 167–180, 1990.
12. B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 455–494. MIT Press, 2000.
13. G. D. Plotkin. LCF considered as a programming language. *Theor. Comput. Sci.*, 5(3):223–255, 1977.
14. P. Sewell, P. T. Wojciechowski, and A. Unyapoth. Nomadic Pict: Programming languages, communication infrastructure overlays, and semantics for mobile computation. *ACM Trans. Program. Lang. Syst.*, 32(4), 2010.
15. M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core.* MIT Press, Cambridge, MA, USA, 2nd. (revised) edition, 1998.
16. A. M. White. A high-level framework for network-based resource sharing. Augmentation Research Centre, 1975. RFC 707.