Higher-Order Concurrent Programs with Finite Communication Topology

(Extended Abstract)

Hanne Riis Nielson Flemming Nielson

Computer Science Department
Aarhus University, Denmark.

e-mail:{hrnielson,fnielson}@daimi.aau.dk

Abstract

Concurrent ML (CML) is an extension of the functional language Standard ML (SML) with primitives for the dynamic creation of processes and channels and for the communication of values over channels. Because of the powerful abstraction mechanisms the communication topology of a given program may be very complex and therefore an efficient implementation may be facilitated by knowledge of the topology.

This paper presents an analysis for determining when a bounded number of processes and channels will be generated. The analysis proceeds in two stages. First we extend a polymorphic type system for SML to deduce not only the type of CML programs but also their communication behaviour expressed as terms in a new process algebra. Next we develop an analysis that given the communication behaviour predicts the number of processes and channels required during the execution of the CML program. The correctness of the analysis is proved using a subject reduction property for the type system.

Appeared in the Proccedings of the 21'st annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 84–97, ACM Press, 1994.

1 Introduction

Higher-order concurrent languages as CML [8] and Facile [2] offer mechanisms for the *dynamic* creation of channels and processes in addition to the possibility of sending and receiving values over channels. To obtain an *efficient implementation* of programs in such languages we would need information about their communication topology:

- Does the program only spawn a finite number of processes?
- Does the program only create a finite number of channels?

If the answer to the first question is yes we may load the processes on the available processors and dispense with *multitasking*. If additionally the answer to the second question is yes we may allocate the channels statically and dispense with *multiplexing*. This leads to considerable savings in the run-time system.

In this paper we present an *analysis of the communication topology* of CML programs. It proceeds in two stages:

- Extract the communication behaviour of the program.
- Analyse the topology of the communication behaviour.

The *first stage* is based on the effect systems developed in [3, 10] for polymorphic type inference of functional

languages with references. The basic insight is that a precise picture of the communication topology necessitates more *causality* in the effects than would result from a straightforward modification of [10] (as done in [12]). We then establish a *subject reduction property* for the effect system and this shows that the execution of the CML program is correctly modelled by the evolution of the communication behaviour.

For the *second stage* we present an inference system that predicts the number of processes and channels created during the evolution of behaviours. The analysis is proved correct with respect to the semantics of behaviours; and using the subject reduction property the correctness result carries over to CML programs.

The idea of using *types* to extract the communication behaviour of programs goes back to [6] and the idea of using *types and effects* to extract the side effects of programs goes back to [3]. An application of types and effects to languages with concurrency constructs includes [12] and this may be regarded as an improvement over [6]. In [7] the approach is extended to retain the *causality* of the various effects and this amounts to extracting a *process algebra* for a mono-typed version of CML. Sections 2, 3 and 4 of the present paper can be viewed as an *extension* of this method to a polymorphically typed version of CML: we define the process algebra of CML and its semantics and then the subject reduction property expresses the relationship between CML and the process algebra.

However, there are also some important differences between the present approach and that of [7]. One is that [7] uses a type system based on subtyping whereas we use instantiation of polymorphic types. Another difference is that here the semantics of behaviours is defined independently of the type system: using a notion of simulation close to that of CHOCS [11] (and CCS [4]) we provide laws for those behaviours that can be used in the type system; this is an improvement over [7] where the laws used in the type system are also used in the specification of the semantics of the behaviours.

In summary, we believe that this paper presents the first provenly correct static analysis of the communication topology of concurrent programs.

2 Polymorphic Behaviours

We shall follow [1, 9] and study a polymorphic subset of CML with expressions $e \in \mathbf{Exp}$ given by

$$\begin{array}{rcl} e & ::= & c \mid x \mid \texttt{fn}\ x\ \texttt{=>}\ e \mid e_1\ e_2 \\ & \mid & \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \mid \texttt{rec}\ f\ x\ \texttt{=>}\ e \\ & \mid & \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \end{array}$$

Here x and f are program variables. In addition to function abstraction and function application we have a polymorphic `let`-construct, recursion and a conditional. The constants $c \in \mathbf{Const}$ are given by

$$\begin{array}{rcl} c & ::= & () \mid \texttt{true} \mid \texttt{false} \mid n \\ & \mid & \texttt{pair} \mid \texttt{fst} \mid \texttt{snd} \\ & \mid & \texttt{nil} \mid \texttt{cons} \mid \texttt{hd} \mid \texttt{tl} \mid \texttt{isnil} \\ & \mid & \texttt{send} \mid \texttt{receive} \mid \texttt{choose} \\ & \mid & \texttt{wrap} \mid \texttt{sync} \mid \texttt{channel} \mid \texttt{fork} \\ & \mid & \texttt{+} \mid \texttt{*} \mid \texttt{=} \mid \cdots \end{array}$$

We have constants corresponding to the base types `unit`, `bool` and `int` together with operations for constructing and destructing pairs and lists. There are operations for sending and receiving values over channels, for choosing between various communication possibilities and for modifying values being communicated. Actually, these primitives construct suspended communications that may be enacted using synchronisation. Finally, there are primitives for creating new channels and processes.

As usual we shall use *types* to classify the values that expressions can evaluate to. When executing a CML program channels and processes may be created and values may be communicated and we shall extend the type system with *behaviours* to record this. Channels are associated with channel identifiers when programs execute and to record this we introduce *regions* into the type system.

For types $t \in \mathbf{Typ}$ we take

$$\begin{array}{rcl} t & ::= & \texttt{unit} \mid \texttt{bool} \mid \texttt{int} \mid \alpha \\ & \mid & t_1 \rightarrow^b t_2 \mid t_1 \times t_2 \mid t\ \texttt{list} \\ & \mid & t\ \texttt{chan}\ r \mid t\ \texttt{com}\ b \end{array}$$

where α is a meta-variable for type variables. The function type is written $t_1 \rightarrow^b t_2$ indicating that the argument type of the functions is t_1, the result type is t_2 and the latent behaviour is b — thus when a function is supplied with an argument the resulting behaviour will be b. The type of a channel is t `chan` r indicating that the channel is allocated in region r and that values of type t can be communicated over it. Finally, t `com` b is the type of a *suspended* communication: when it eventually is enacted using `sync`, it will result in a value of type t and the resulting behaviour will be b.

Formally, behaviours $b \in \mathbf{Beh}$ are given by

$$\begin{array}{rcl} b & ::= & \epsilon \mid r!t \mid r?t \mid t\ \textsc{chan}\ r \mid \beta \\ & \mid & \textsc{fork}\ b \mid b_1; b_2 \mid b_1 + b_2 \mid \textsc{rec}\ \beta.b \end{array}$$

Here ϵ stands for the non-observable behaviour. We write $r!t$ for sending a value of type t over a channel

c	TypeOf(c)
`()`	`unit`
`true`	`bool`
`false`	`bool`
n	`int`
`pair`	$\forall\alpha_1,\alpha_2,\beta_1,\beta_2.\ \alpha_1 \rightarrow^{\epsilon+\beta_1} \alpha_2 \rightarrow^{\epsilon+\beta_2} \alpha_1 \times \alpha_2$
`fst`	$\forall\alpha_1,\alpha_2,\beta.\ \alpha_1 \times \alpha_2 \rightarrow^{\epsilon+\beta} \alpha_1$
`snd`	$\forall\alpha_1,\alpha_2,\beta.\ \alpha_1 \times \alpha_2 \rightarrow^{\epsilon+\beta} \alpha_2$
`nil`	$\forall\alpha.\ \alpha$ `list`
`cons`	$\forall\alpha,\beta_1,\beta_2.\ \alpha \rightarrow^{\epsilon+\beta_1} \alpha$ `list` $\rightarrow^{\epsilon+\beta_2} \alpha$ `list`
`hd`	$\forall\alpha,\beta.\ \alpha$ `list` $\rightarrow^{\epsilon+\beta} \alpha$
`tl`	$\forall\alpha,\beta.\ \alpha$ `list` $\rightarrow^{\epsilon+\beta} \alpha$ `list`
`isnil`	$\forall\alpha,\beta.\ \alpha$ `list` $\rightarrow^{\epsilon+\beta}$ `bool`
`send`	$\forall\alpha,\beta_1,\beta_2,\rho.\ (\alpha$ `chan` $\rho) \times \alpha \rightarrow^{\epsilon+\beta_1} \alpha$ `com` $(\rho!\alpha+\beta_2)$
`receive`	$\forall\alpha,\beta_1,\beta_2,\rho.\ (\alpha$ `chan` $\rho) \rightarrow^{\epsilon+\beta_1} \alpha$ `com` $(\rho?\alpha+\beta_2)$
`choose`	$\forall\alpha,\beta_1,\beta_2,\beta_3.\ (\alpha$ `com` $\beta_1)$ `list` $\rightarrow^{\epsilon+\beta_2} \alpha$ `com` $(\beta_1+\beta_3)$
`wrap`	$\forall\alpha_1,\alpha_2,\beta_1,\beta_2,\beta_3,\beta_4.\ (\alpha_1$ `com` $\beta_1) \times (\alpha_1 \rightarrow^{\beta_2} \alpha_2) \rightarrow^{\epsilon+\beta_3} \alpha_2$ `com` $((\beta_1;\beta_2)+\beta_4)$
`sync`	$\forall\alpha,\beta_1,\beta_2.\ (\alpha$ `com` $\beta_1) \rightarrow^{\beta_1+\beta_2} \alpha$
`channel`	$\forall\alpha,\beta,\rho.$ `unit` $\rightarrow^{(\alpha\ \textsc{chan}\ \rho)+\beta} (\alpha$ `chan` $\rho)$
`fork`	$\forall\alpha,\beta_1,\beta_2.\ ($`unit` $\rightarrow^{\beta_1} \alpha) \rightarrow^{(\textsc{fork}\ \beta_1)+\beta_2}$ `unit`
`+`	$\forall\beta_1,\beta_2.$ `int` $\rightarrow^{\epsilon+\beta_1}$ `int` $\rightarrow^{\epsilon+\beta_2}$ `int`
$\vdots$	$\vdots$

Table 1: Type Schemes for Constants

in region r and similarly $r?t$ for receiving a value of type t over a channel in region r. The allocation of a new channel in region r is written t CHAN r where t is the type of values to be communicated. The behaviour FORK b expresses that a process with behaviour b is spawned. Behaviours may be combined using sequencing and choice and they may be recursive. We write β for a meta-variable for behaviour variables. So for example REC β. $(t$ CHAN $r + \textsc{fork}(r?t;\ \beta))$ is the behaviour of a program that either will create a channel and then no more communications take place or it will spawn a process that inputs on some channel and then the overall process will recurse.

Finally, regions $r \in \mathbf{Reg}$ are given by

$$\begin{array}{rcl} r & ::= & \mathtt{r}_0 \mid \mathtt{r}_1 \mid \cdots \\ & \mid & \rho \end{array}$$

Here ρ denotes a meta-variable for region variables and $\mathtt{r}_0$, $\mathtt{r}_1$, ... denote region constants.

The *type schemes* are obtained from types by quantifying over type variables, behaviour variables and region variables: they have the form $\forall\vec{\alpha}\vec{\beta}\vec{\rho}.t$ where $\vec{\alpha}$, $\vec{\beta}$ and $\vec{\rho}$ are lists of variables. Each constant has associated a type scheme as shown in Table 1.

Occasionally, the context may demand that a subexpression is given a type with a latent behaviour that is larger than it really is. This is illustrated by the following example:

Example 1 Consider the program

```
choose [send (ch, 7),
        wrap (receive ch', fn x => 1)]
```

$tenv \vdash c : t \;\&\; b$	if $\mathrm{TypeOf}(c) \succ t$ and $\epsilon \sqsubseteq b$
$tenv \vdash x : t \;\&\; b$	if $tenv(x) \succ t$ and $\epsilon \sqsubseteq b$
$\dfrac{tenv[x \mapsto t] \vdash e : t' \;\&\; b}{tenv \vdash \texttt{fn}\ x\ \texttt{=>}\ e : t \rightarrow^b t' \;\&\; b'}$	if $\epsilon \sqsubseteq b'$
$\dfrac{tenv \vdash e_1 : t \rightarrow^b t' \;\&\; b_1 \quad tenv \vdash e_2 : t \;\&\; b_2}{tenv \vdash e_1\ e_2 : t' \;\&\; b'}$	if $b_1; b_2; b \sqsubseteq b'$
$\dfrac{tenv \vdash e_1 : t_1 \;\&\; b_1 \quad tenv[x \mapsto ts] \vdash e_2 : t_2 \;\&\; b_2}{tenv \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : t_2 \;\&\; b'}$	if $ts = \mathrm{gen}(tenv, b_1)t_1$ and $b_1; b_2 \sqsubseteq b'$
$\dfrac{tenv[f \mapsto t \rightarrow^b t'][x \mapsto t] \vdash e : t' \;\&\; b}{tenv \vdash \texttt{rec}\ f(x)\texttt{=>}e : t \rightarrow^b t' \;\&\; b'}$	if $\epsilon \sqsubseteq b'$
$\dfrac{tenv \vdash e : \texttt{bool} \;\&\; b \quad tenv \vdash e_1 : t \;\&\; b_1 \quad tenv \vdash e_2 : t \;\&\; b_2}{tenv \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : t \;\&\; b'}$	if $b; (b_1 + b_2) \sqsubseteq b'$

Table 2: Typing System

where we for the sake of readability write `[`e_1`,` e_2`]` for `cons` e_1 `(cons` e_2 `nil)` and `(`e_1`,` e_2`)` for `pair` e_1 e_2. The first element of the list has type

```
int com r!int
```

(assuming `ch` has type `int chan r`) and the second element has type

```
int com r'?bool
```

(assuming `ch'` has type `bool chan r'`). We want the list to have type

```
int com (r!int + r'?bool)
```

to record that either one of the branches may be chosen at run-time. So we need to coerce the types `int com r!int` and `int com r'?bool` into `int com (r!int + r'?bool)`. □

Basically, there are two approaches we may adopt:

- *Late subsumption*: coercions can happen at any time inside any type, as when the type system has a general subsumption rule on types.
- *Early subsumption*: generic instantiations produce the required specialised types.

In [7] we used the first approach for a mono-typed version of the language thereby obtaining a type system with subtyping. We are now in a polymorphic setting and to avoid potential problems from the presence of both polymorphism and subtyping we shall use the second approach (also taken in [10]). This means that the latent behaviour of functions and suspended communications always must be prepared to be larger than what seems to be needed and this accounts for the "$+\beta$" components in the behaviours of the form $b + \beta$ in Table 1. Note that only the constants `sync`, `channel` and `fork` have a non-trivial latent behaviour (i.e. have $b \neq \epsilon$).

A type t is a *generic instance* of a type scheme $ts = \forall\vec{\alpha}\vec{\beta}\vec{\rho}.t_0$, written

$$ts \succ t$$

if there exists a substitution θ with $\mathrm{Dom}(\theta) = \{\vec{\alpha}\vec{\beta}\vec{\rho}\}$ such that $\theta\ t_0 = t$. Here a *substitution* θ is a finite mapping from type variables, behaviour variables and region variables to types, behaviours and regions, respectively, and we write $\mathrm{Dom}(\theta)$ for its (finite) domain. Furthermore, a type scheme ts' is an instance of ts, written

$$ts \succ ts'$$

if whenever $ts' \succ t$ also $ts \succ t$.

The *typing judgements* have the form

$$tenv \vdash e : t \;\&\; b$$

where $tenv$ is a type environment mapping variables to type schemes, t is the type of e and b is its behaviour. Since CML has a call-by-value semantics there is no behaviour associated with accessing a variable and therefore the type environment does not contain any behaviour component (except within type schemes). The typing rules are given in Table 2 and are fairly close to the standard ones with the only exception being that also behaviour information is collected. The types

• pre-order laws	**P1.** $b \sqsubseteq b$
	P2. if $b_1 \sqsubseteq b_2$ and $b_2 \sqsubseteq b_3$ then $b_1 \sqsubseteq b_3$
• pre-congruence laws	**C1.** if $b_1 \sqsubseteq b_2$ and $b_3 \sqsubseteq b_4$ then $b_1;b_3 \sqsubseteq b_2;b_4$
	C2. if $b_1 \sqsubseteq b_2$ and $b_3 \sqsubseteq b_4$ then $b_1 + b_3 \sqsubseteq b_2 + b_4$
	C3. if $b_1 \sqsubseteq b_2$ then FORK $b_1 \sqsubseteq$ FORK b_2
	C4. if $b_1 \sqsubseteq b_2$ then REC $\beta.\ b_1 \sqsubseteq$ REC $\beta.\ b_2$
• laws for sequencing	**S1.** $b_1;(b_2;b_3) \sqsubseteq (b_1;b_2);b_3$ and $(b_1;b_2);b_3 \sqsubseteq b_1;(b_2;b_3)$
	S2. $(b_1 + b_2);b_3 \sqsubseteq (b_1;b_3) + (b_2;b_3)$ and $(b_1;b_3) + (b_2;b_3) \sqsubseteq (b_1 + b_2);b_3$
• laws for ϵ	**E1.** $b \sqsubseteq \epsilon;b$ and $\epsilon;b \sqsubseteq b$
	E2. $b;\epsilon \sqsubseteq b$ and $b \sqsubseteq b;\epsilon$
• laws for choice (or join)	**J1.** $b_1 \sqsubseteq b_1 + b_2$ and $b_2 \sqsubseteq b_1 + b_2$
	J2. $b + b \sqsubseteq b$
• laws for recursion	**R1.** REC $\beta.\ b \sqsubseteq b[\beta \mapsto$ REC $\beta.\ b]$ and $b[\beta \mapsto$ REC $\beta.\ b] \sqsubseteq$ REC $\beta.\ b$

Table 3: Ordering on Behaviours

of constants and variables are obtained as generic instances of the appropriate type schemes. The actual behaviour is ϵ but, as mentioned earlier, we may want to use a larger behaviour and to express this we introduce an ordering $\sqsubseteq$ on behaviours. This turns out to be a general pattern of the axioms and rules: it is always possible to enlarge the actual behaviour.

In the rule for function abstraction we record the behaviour of the body of the function as the latent behaviour of the function type. The construction of a function does not in itself have an observable behaviour and so is ϵ. In the rule for function application we see that the actual behaviour of the composite construct is that of the operator followed by that of the operand and then the behaviour initiated by the function application itself; the latter is exactly the latent behaviour of the function type. One may note that it is inherent in this rule that CML has a call-by-value semantics. In the rule for local definitions we generalise over those type variables, behaviour variables and region variables that neither occur free in the type environment nor in the behaviour; this is expressed by

$$\begin{array}{l}\mathrm{gen}(tenv, b)t = \\ \quad \text{let } \{\vec{\alpha}\vec{\beta}\vec{\rho}\} = FV(t) \setminus (FV(tenv) \cup FV(b)) \\ \quad \text{in } \forall\vec{\alpha}\vec{\beta}\vec{\rho}.t\end{array}$$

where FV$(\cdots)$ is the set of free type variables, behaviour variables and region variables of $\cdots$. The actual behaviour of the `let`-construct simply expresses that the local value is computed before the body. In the rule for recursive functions we make sure that the actual behaviour is equal to the latent behaviour of the type of the recursive function. The rule for conditional should be straightforward.

The ordering $\sqsubseteq$ on behaviours is defined by the axioms and rules of Table 3. Thus we require $\sqsubseteq$ to be a pre-order and a pre-congruence. Furthermore, sequencing is an associative operation with ϵ as identity and we have a distributive law with respect to choice. A consequence of the laws for choice is that choice is associative and commutative. Finally, the law for recursion allows to unfold the REC-construct.

Remark The main difference between the typing system presented here and those of [10, 12] is that we keep the dependencies between the individual communications. If we were to extend Table 3 with

$$\begin{array}{ll} b_1;b_2 \sqsubseteq b_1 + b_2 & b_1 + b_2 \sqsubseteq b_1;b_2 \\ \text{REC } \beta.\ b \sqsubseteq b[\beta \mapsto \epsilon] & b[\beta \mapsto \epsilon] \sqsubseteq \text{REC } \beta.\ b \end{array}$$

then our system would degrade to the level of the systems in [10, 12]. □

Example 2 To illustrate the kind of information provided by the behaviours consider the following program. It is given a list of functions and two channels and then creates one process for each function in the list and subsequently connects all functions into a pipeline.

```
let process =
    fn f => fn in => fn out =>
       fork (rec loop d =>
             sync (wrap (receive in,
                   fn x =>
                   sync (send (out, f x))));
             loop())
in rec pipe fs => fn in => fn out =>
       if isnil fs
       then process (fn x => x) in out
       else let ch = channel ()
            in (process (hd fs) in ch;
                pipe (tl fs) ch out)
```

For the sake of readability we have written $e_1; e_2$ as an abbreviation for (`fn dummy =>` e_2) e_1. The type of `process` is

$$\forall\alpha_1,\alpha_2,\beta,\rho_1,\rho_2.\ (\alpha_1 \to^{\beta} \alpha_2) \to^{\epsilon} (\alpha_1\ \texttt{chan}\ \rho_1) \to^{\epsilon} (\alpha_2\ \texttt{chan}\ \rho_2) \to^{b} \texttt{unit}$$

where

$$b = \textsc{fork}\ (\textsc{rec}\ \beta'.\ (\rho_1?\alpha_1;\ \beta;\ \rho_2!\alpha_2;\ \beta'))$$

From this behaviour we see that `process` will create one process and no channels. Turning to the main program the type is

$$\forall\alpha,\beta,\rho.\ (\alpha \to^{\beta} \alpha)\ \texttt{list} \to^{\epsilon} (\alpha\ \texttt{chan}\ \rho) \to^{\epsilon} (\alpha\ \texttt{chan}\ \rho) \to^{b'} \texttt{unit}$$

where

$$b' = \textsc{rec}\ \beta'.\ (\textsc{fork}\ (\textsc{rec}\ \beta''.\ (\rho?\alpha;\ \rho!\alpha;\ \beta'')) + \alpha\ \textsc{chan}\ \rho;\ \textsc{fork}\ (\textsc{rec}\ \beta''.\ (\rho?\alpha;\ \beta;\ \rho!\alpha;\ \beta''));\ \beta')$$

From this we see that any number of processes and any number of channels may be generated. □

3 Semantics

We shall now present a structural operational semantics for CML. The formulation is close in spirit to [9] and amounts to three inference systems: one for *sequential* evaluation, one for *concurrent* evaluation, and to handle synchronisation we also need one for *matching* the communications against one another. This is mimicked in the specification of the semantics of behaviours where we have one inference system for *sequential* evolution and one for *concurrent* evolution. Matching is much simpler for behaviours than for programs so there is no need for a special matching relation.

Operator	Operand	Result
`pair`	w_1	$\langle$`pair` $w_1\rangle$
$\langle$`pair` $w_1\rangle$	w_2	$\langle$`pair` $w_1\, w_2\rangle$
`fst`	$\langle$`pair` $w_1\, w_2\rangle$	w_1
`snd`	$\langle$`pair` $w_1\, w_2\rangle$	w_2
`cons`	w_1	$\langle$`cons` $w_1\rangle$
$\langle$`cons` $w_1\rangle$	w_2	$\langle$`cons` $w_1\, w_2\rangle$
`hd`	$\langle$`cons` $w_1\, w_2\rangle$	w_1
`tl`	$\langle$`cons` $w_1\, w_2\rangle$	w_2
`isnil`	`nil`	`true`
`isnil`	$\langle$`cons` $w_1\, w_2\rangle$	`false`
`send`	w	$\langle$`send` $w\rangle$
`receive`	w	$\langle$`receive` $w\rangle$
`choose`	w	$\langle$`choose` $w\rangle$
`wrap`	w	$\langle$`wrap` $w\rangle$
`+`	n_1	$\langle$`+` $n_1\rangle$
$\langle$`+` $n_1\rangle$	n_2	n where $n = n_1 + n_2$
⋮	⋮	⋮

Table 4: Tabulation of δ

Semantics of CML

We begin with the *sequential evaluation* of expressions. This takes care of all primitives of CML except `sync`, `channel` and `fork` which are the constants of Table 1 with a non-trivial latent behaviour. The transition relation for sequential evaluation has the form

$$e \to e'$$

where e and e' are *closed* expressions, i.e. they do not contain free program variables. To enforce a left-to-right evaluation we introduce the concept of an *evaluation context* E which specifies where the next step of the computation may take place:

$$E ::= [\,] \mid E\,e \mid w\,E \mid \texttt{let}\ x = E\ \texttt{in}\ e \mid \texttt{if}\ E\ \texttt{then}\ e_1\ \texttt{else}\ e_2$$

$$
\begin{array}{rcll}
E\,[\texttt{rec}\ f(x)\ \texttt{=>}\ e] & \rightarrow & E\,[\,(\texttt{fn}\ x\ \texttt{=>}\ e)\ [f \mapsto (\texttt{rec}\ f(x)\ \texttt{=>}\ e)]] & \\
E\,[\,(\texttt{fn}\ x\ \texttt{=>}\ e)\ w\,] & \rightarrow & E\,[e\ [x \mapsto w]] & \\
E\,[\texttt{let}\ x = w\ \texttt{in}\ e\,] & \rightarrow & E\,[e\ [x \mapsto w]] & \\
E\,[\texttt{if}\ w\ \texttt{then}\ e_1\ \texttt{else}\ e_2] & \rightarrow & \left\{ \begin{array}{ll} E\,[e_1] & \text{if } w = \texttt{true} \\ E\,[e_2] & \text{if } w = \texttt{false} \end{array} \right. & \\
E\,[w_1\,w_2] & \rightarrow & E\,[w_3] & \text{if } (w_1, w_2, w_3) \in \delta
\end{array}
$$

Table 5: Sequential Evaluation

Here $w \in \mathbf{WExp}$ denotes a *weakly evaluated expression*, i.e. an expression that cannot be further evaluated. The idea is that [] is an empty context (called a hole) and in general E specifies a context with exactly one hole in it. We shall then write $E[e]$ for the expression E with the hole replaced by e. The next step of the computation will take place at the point indicated by the hole. As an example consider function application. The presence of $E\,e$ means that computations in the operator position are possible whereas the presence of $w\,E$ means that computations in the operand position only are possible when the operator is weakly evaluated (e.g. to a function abstraction). In this way it is ensured that the operator as well as the operand are evaluated before the function application itself takes place (e.g. by a β-reduction).

The weakly evaluated expressions are given by

$$
\begin{array}{rcl}
w & ::= & c \mid x \mid \texttt{fn}\ x\ \texttt{=>}\ e \\
& \mid & \langle c\,w_1\rangle \mid \ldots \mid \langle c\,w_1 \ldots w_n\rangle
\end{array}
$$

where $n \geq 1$. Weakly evaluated expressions of the form $\langle c\,w_1 \ldots w_i\rangle$ are used to record the evaluation of constants as indicated in Table 4 where we define the relation $\delta \subseteq \mathbf{WExp} \times \mathbf{WExp} \times \mathbf{WExp}$.

The transition relation is specified in Table 5 where we write $e[x \mapsto w]$ for the expression e with all free occurrences of x replaced by w. The clauses should be fairly straightforward. The first rule expresses the one-level unfolding of a recursive definition. Then we have axioms for β-reduction and for `let`-reduction. The fourth axiom is an abbreviation for two axioms expressing the evaluation of a conditional depending on the outcome of the test. Finally, there is an axiom for δ-reduction which inspects Table 4 to determine the result.

We shall now introduce the transition relation for *concurrent evaluation*. Channels will be associated with *channel identifiers*, $ci \in \mathbf{CIdent}$, and processes with *process identifiers*, $pi \in \mathbf{PIdent}$. The configurations have the form $CI\ \&\ PP$ where CI is the set of channel identifiers that are in use and PP is a (finite) mapping of process identifiers to expressions. The transition relation is written

$$CI\ \&\ PP \longrightarrow_{ps}^{ev} CI'\ \&\ PP'$$

where ev is the event that takes place and ps is a list of the processes that take part in the event – depending on the event there will be either one or two processes involved. An *event* has one of the forms

$$ev ::= \epsilon \mid \textsc{chan}\ ci \mid \textsc{fork}\ pi \mid (ci!, ci?)$$

and may record the empty event, the creation of a channel with a given channel identifier, the creation of a process with a given process identifier and the communication over a channel.

The transition relation is specified in Table 6. The first rule embeds sequential evaluation within concurrent evaluation and the name of the process performing the event is recorded. The second rule captures the creation of a new channel. The channel is associated with a new channel identifier and the transition records the name of the process performing the event together with the event itself. The third rule takes care of process creation and follows the same lines. Here we record the process performing the event as well as the one being created by the event. Finally, we have a rule expressing the synchronisation of communications and here we use the matching relation. The transition records the two processes involved in the communication as well as the channel used for it.

Finally, the *matching relation* is given two weakly evaluated expressions that are ready to synchronise and it specifies the outcome of the communication and records the event that takes place. This is expressed by a relation of the form

$$(w_1, w_2) \overset{(ci!,ci?)}{\rightsquigarrow} (e_1, e_2) \text{ and}$$
$$(w_1, w_2) \overset{(ci?,ci!)}{\rightsquigarrow} (e_1, e_2)$$

The relation is specified in Table 7. The first axiom captures the communication between a `send` and a `receive`

$$\frac{E[e] \rightarrow E[e']}{CI \,\&\, PP[pi \mapsto E[e]] \longrightarrow^{\epsilon}_{pi} CI \,\&\, PP[pi \mapsto E[e']]}$$

$$CI \,\&\, PP[pi \mapsto E[\texttt{channel ()}]] \longrightarrow^{\text{CHAN } ci}_{pi} CI \cup \{ci\} \,\&\, PP[pi \mapsto E[ci]] \qquad \text{if } ci \notin CI$$

$$CI \,\&\, PP[pi_1 \mapsto E[\texttt{fork } w]] \longrightarrow^{\text{FORK } pi_2}_{pi_1,pi_2} CI \,\&\, PP[pi_1 \mapsto E[()]][pi_2 \mapsto w\,()] \qquad \text{if } pi_2 \notin \text{Dom}(PP) \cup \{pi_1\}$$

$$\frac{(w_1, w_2) \overset{(ci!,ci?)}{\rightsquigarrow} (e_1, e_2)}{CI \,\&\, PP[pi_1 \mapsto E_1[\texttt{sync } w_1]][pi_2 \mapsto E_2[\texttt{sync } w_2]] \longrightarrow^{(ci!,ci?)}_{pi_1,pi_2} CI \,\&\, PP[pi_1 \mapsto E_1[e_1]][pi_2 \mapsto E_2[e_2]]} \qquad \text{if } pi_1 \neq pi_2$$

Table 6: Concurrent Evaluation

$$(\langle \texttt{send}\langle \texttt{pair } ci\ w\rangle\rangle, \langle \texttt{receive } ci\rangle) \overset{(ci!,ci?)}{\rightsquigarrow} (w, w)$$

$$\frac{(w_1, w_3) \overset{(d_1,d_2)}{\rightsquigarrow} (e_1, e_3)}{(\langle \texttt{choose}\langle \texttt{cons } w_1\ w_2\rangle\rangle, w_3) \overset{(d_1,d_2)}{\rightsquigarrow} (e_1, e_3)}$$

$$\frac{(\langle \texttt{choose } w_2\rangle, w_3) \overset{(d_1,d_2)}{\rightsquigarrow} (e_2, e_3)}{(\langle \texttt{choose}\langle \texttt{cons } w_1\ w_2\rangle\rangle, w_3) \overset{(d_1,d_2)}{\rightsquigarrow} (e_2, e_3)}$$

$$\frac{(w_1, w_3) \overset{(d_1,d_2)}{\rightsquigarrow} (e_1, e_3)}{(\langle \texttt{wrap}\langle \texttt{pair } w_1\ w_2\rangle\rangle, w_3) \overset{(d_1,d_2)}{\rightsquigarrow} (w_2\, e_1, e_3)}$$

$$\frac{(w_1, w_2) \overset{(d_1,d_2)}{\rightsquigarrow} (e_1, e_2)}{(w_2, w_1) \overset{(d_2,d_1)}{\rightsquigarrow} (e_2, e_1)}$$

Table 7: Matching

construct. The second and third axiom take care of the situation where there are several possible communications available in the first component. The fourth axiom shows how the communicated value may be modified using the `wrap` construct. Finally, we have a restructuring rule.

Semantics of Behaviours

We begin with the *sequential* evolution of behaviours. Here the configurations of the transition system are either *closed* behaviours, i.e. behaviours without free behaviour variables, or the special terminating configuration $\surd$. The transition relation takes the form

$$b \Rightarrow^{p} \flat$$

where $\flat$ is either a closed behaviour or $\surd$, and where p is an *atomic behaviour* as given by

$$p \quad ::= \quad \epsilon \mid r!t \mid r?t \mid t \text{ CHAN } r \mid \text{FORK } b$$

Here ϵ is supposed to capture the sequential evaluation steps of CML expressions whereas the remaining atomic behaviours capture the concurrent steps.

The relation is specified in Table 8. The first axiom expresses that any atomic behaviour can be performed and in doing so becomes ϵ. The second axiom expresses that ϵ can terminate; in fact this is the only behaviour that in one step can reach the terminal configuration. The third axiom means that at any time any number of ϵ actions can be performed by any behaviour (observe that the terminal configuration is excluded here – $\surd$ is a stuck configuration). This axiom reflects that in the semantics of CML any number of evaluation steps can be performed in the functional part of CML between those involving the concurrency primitives. The fourth axiom expresses the unfolding of a recursive behaviour. Then we have two rules for the evolution of sequential behaviours: only when the evolution of the first component has reached a terminal configuration is it possible to start evolution of the second component. The last two rules express the evolution of a choice between two behaviours.

To express the *concurrent evolution* of behaviours we introduce *process identifiers* as in the semantics of CML. The transitions have the form

$$PB \Longrightarrow^{a}_{ps} PB'$$

where PB and PB' are mappings from process identifiers to closed behaviours and the special symbol $\surd$. Furthermore, a is the action that takes place and ps is a list of the processes that take part in the action. As in the semantics of CML, ps has one or two elements depending on the action. The *actions* are given by

$$a ::= \epsilon \mid t \text{ CHAN } r \mid \text{FORK } b \mid (r!t, r?t)$$

The transition relation is specified in Table 9. The first four rules embed sequential evolution in the concurrent

$$\frac{b \Rightarrow^{\epsilon} \surd}{PB[pi \mapsto b] \Longrightarrow^{\epsilon}_{pi} PB[pi \mapsto \surd]}$$

$$\frac{b \Rightarrow^{\epsilon} b'}{PB[pi \mapsto b] \Longrightarrow^{\epsilon}_{pi} PB[pi \mapsto b']}$$

$$\frac{b \Rightarrow^{t\ \mathrm{CHAN}\ r} b'}{PB[pi \mapsto b] \Longrightarrow^{t\ \mathrm{CHAN}\ r}_{pi} PB[pi \mapsto b']}$$

$$\frac{b \Rightarrow^{\mathrm{FORK}\ b_0} b'}{PB[pi_1 \mapsto b] \Longrightarrow^{\mathrm{FORK}\ b_0}_{pi_1, pi_2} PB[pi_1 \mapsto b'][pi_2 \mapsto b_0]} \quad \text{if } pi_2 \notin \mathrm{Dom}(PB) \cup \{pi_1\}$$

$$\frac{b_1 \Rightarrow^{r!t} b_1' \qquad b_2 \Rightarrow^{r?t} b_2'}{PB[pi_1 \mapsto b_1][pi_2 \mapsto b_2] \Longrightarrow^{(r!t, r?t)}_{pi_1, pi_2} PB[pi_1 \mapsto b_1'][pi_2 \mapsto b_2']} \quad \text{if } pi_1 \neq pi_2$$

Table 9: Concurrent Evolution

$p \Rightarrow^{p} \epsilon$	$\epsilon \Rightarrow^{\epsilon} \surd$
$b \Rightarrow^{\epsilon} b$	$\mathrm{REC}\ \beta.\ b \Rightarrow^{\epsilon} b[\beta \mapsto \mathrm{REC}\ \beta.\ b]$
$\dfrac{b_1 \Rightarrow^{p} b_1'}{b_1; b_2 \Rightarrow^{p} b_1'; b_2}$	$\dfrac{b_1 \Rightarrow^{p} \surd}{b_1; b_2 \Rightarrow^{p} b_2}$
$\dfrac{b_1 \Rightarrow^{p} b_1'}{b_1 + b_2 \Rightarrow^{p} b_1'}$	$\dfrac{b_2 \Rightarrow^{p} b_2'}{b_1 + b_2 \Rightarrow^{p} b_2'}$

Table 8: Sequential Evolution

evolution: the first rule captures the termination of a behaviour, the second rule captures when the behaviour has a trivial atomic behaviour, the third rule when a channel is created and the fourth rule when a process is created. In all cases the action as well as the processes involved are recorded. The final rule captures the communication between processes. Here the matching simply amounts to ensuring that the channels of the two processes are in the same region and that they agree on the type of the value being communicated.

4 Subject Reduction Property

We shall prove that the typing system of Section 2 has the following subject reduction properties:

- Types are preserved during computation.
- Behaviours evolve during computation.

The formalisation and proof of this result is in three stages: First, we prove a subject reduction property for the sequential evaluation of expressions. Then we prove a correctness property for matching and finally we prove the subject reduction property for concurrent evaluation. The proof of the latter involves showing that the ordering $\sqsubseteq$ on behaviours as defined in Table 3 is sound with respect to a simulation ordering obtained from the semantics of behaviours.

For sequential correctness it is natural to restrict attention to closed expressions because the definition of an evaluation context is such that we never pass inside the scope of any defining occurrence of a program variable. However, we have to allow expressions to include channel identifiers that have been allocated during the computation. To formalise this we shall write *cenv* for a mapping from channel identifiers to types (so *cenv ci* will always have the form t `chan` r). We shall say that e is *closed* if $cenv \vdash e : t \ \& \ b$ for some *cenv*, t and b. To express the correctness result we shall also need typing rules for weakly evaluated expressions. They have the form

$$\frac{tenv \vdash c : t \rightarrow^{\epsilon} t' \ \& \ \epsilon \quad tenv \vdash w_1 : t \ \& \ \epsilon}{tenv \vdash \langle c\ w_1 \rangle : t' \ \& \ b}$$

if $\epsilon \sqsubseteq b$

$$\frac{tenv \vdash \langle c\ w_1 \cdots w_{n-1} \rangle : t \rightarrow^{\epsilon} t' \ \& \ \epsilon \quad tenv \vdash w_n : t \ \& \ \epsilon}{tenv \vdash \langle c\ w_1\ \cdots\ w_n \rangle : t' \ \& \ b}$$

if $\epsilon \sqsubseteq b$

Here we rely on the fact that whenever $tenv \vdash w : t \ \& \ b$ then also $tenv \vdash w : t \ \& \ \epsilon$. Also we have made it explicit that the latent behaviour of the constants of Table 4 has the form $\epsilon + b'$ and that we always can ignore the b' component. Then we have

Proposition 3 Assume $e \rightarrow e'$ and

$$cenv \vdash e : t \ \& \ b.$$

Then

$$cenv \vdash e' : t \ \& \ b. \qquad \square$$

The proof of Proposition 3 is by induction on the inference $e \rightarrow e'$.

The matching of two weakly evaluated expressions gives rise to a new pair of expressions. To formalise this we shall define $cenv\ ci! = r!t$ and $cenv\ ci? = r?t$ whenever $cenv\ ci = t\ \texttt{chan}\ r$. Then we have

Proposition 4 Assume $(w_1, w_2) \overset{(d_1,d_2)}{\sim\!\!>} (e_1, e_2)$ and

$$cenv \vdash w_1 : t_1\ \texttt{com}\ b_1\ \&\ \epsilon$$
$$cenv \vdash w_2 : t_2\ \texttt{com}\ b_2\ \&\ \epsilon$$

Then there exists b_1' and b_2' such that

$$cenv \vdash e_1 : t_1\ \&\ b_1'$$
$$cenv \vdash e_2 : t_2\ \&\ b_2'$$

and $(cenv\ d_1); b_1' \sqsubseteq b_1$ and $(cenv\ d_2); b_2' \sqsubseteq b_2$. □

The proof is by induction on the inference for matching.

In order to formulate and prove the concurrent correctness result we need to relate the ordering $\sqsubseteq$ on behaviours to the semantics of behaviours. Basically this amounts to the definition of a simulation relation on behaviours and a soundness proof for the laws of Table 3. First define

$$\flat \Rightarrow\!\!\widehat{\ }^{p}\ \flat$$

to mean that there exists behaviours $\flat_1, \cdots, \flat_n$ such that

$$\flat \Rightarrow^{\epsilon} \flat_1 \Rightarrow^{\epsilon} \cdots \Rightarrow^{\epsilon} \flat_n \Rightarrow^{p} \flat$$

Recall that $\flat$ ranges over closed behaviours as well as $\surd$. Thus the atomic behaviour p may be prefixed by any number of trivial atomic behaviours.

We shall say that $\mathcal{S}$ is a *simulation* on (closed) behaviours if

- $\surd\ \mathcal{S}\ \flat$ if and only if $\flat = \surd$
- if $\flat_1 \Rightarrow^{p_1} \flat_1$ and $\flat_1\ \mathcal{S}\ \flat_2$ then there exists $\flat_2$ and p_2 such that $\flat_2 \Rightarrow\!\!\widehat{\ }^{p_2}\ \flat_2$, $p_1\ \mathcal{S}\ p_2$ and $\flat_1\ \mathcal{S}\ \flat_2$.

We define $\sqsubseteq\!\!\!\!\sim$ as the *largest* simulation. The simulation $\sqsubseteq\!\!\!\!\sim$ is extended to *open* behaviours as follows: $b_1 \sqsubseteq\!\!\!\!\sim\ b_2$ holds if for all ground substitutions θ we have $\theta\ b_1 \sqsubseteq\!\!\!\!\sim\ \theta\ b_2$. This definition of simulation is inspired by the notions of bisimilarity as developed for the process algebras CCS [4] and CHOCS [11]. Then we have

Proposition 5 If $b_1 \sqsubseteq b_2$ then $b_1 \sqsubseteq\!\!\!\!\sim\ b_2$. □

The concurrent subject reduction property expresses that each step of the concurrent evaluation of the expression can be mimicked by a number of steps in the concurrent evolution of its behaviour.

Let us first relate the *configurations* $CI\ \&\ PP$ of the concurrent evaluation of expressions to the configurations PB of the concurrent evolution of behaviours. We shall say that $CI\ \&\ PP$ is $cenv$-related to PB if $\mathrm{Dom}(PP) = \mathrm{Dom}(PB)$ and $\mathrm{Dom}(cenv) = CI$. This ensures that we are dealing with the same processes and channel identifiers.

We shall also need to relate the *events* ev of the concurrent evaluation of expressions to the *actions* a of the concurrent evolution of behaviours. So assume that we have two configurations $CI\ \&\ PP$ and PB that are $cenv$-related as explained above. Clearly we would expect FORK pi to correspond to FORK $(PB\ pi)$ and CHAN ci to correspond to t CHAN r when $cenv\ ci = t\ \texttt{chan}\ r$; this is formalised by an auxiliary function denoted $(cenv, PB)$.

The final preparation is to introduce a notation for a sequence of steps in the concurrent evolution of behaviours. For this we write

$$PB \Longrightarrow\!\!\widehat{\ }^{a}_{ps}\ PB'$$

to mean that there exists configurations $PB_1, \cdots, PB_n$ such that

$$PB \Longrightarrow^{\epsilon}_{pi_1} \cdots \Longrightarrow^{\epsilon}_{pi_n} PB_n \Longrightarrow^{a}_{ps} PB'$$

where $pi_1, \cdots, pi_n$ are process identifiers from the list ps. Thus the processes of ps are allowed to perform some trivial actions before they engage in the (joint) action a. We then have

Theorem 6 Assume

$$CI\ \&\ PP \longrightarrow^{ev}_{ps} CI'\ \&\ PP'$$

and let $cenv$ and PB be such that $CI\ \&\ PP$ is $cenv$-related to PB, and for all $pi \in \mathrm{Dom}(PP)$

$$cenv \vdash PP\ pi : t_{pi}\ \&\ PB\ pi$$

for some t_{pi}. Then there exists $cenv'$ and PB' such that

$$PB \Longrightarrow\!\!\widehat{\ }^{a}_{ps}\ PB'$$

where $a = (cenv', PB')ev$ and furthermore $CI'\ \&\ PP'$ is $cenv'$-related to PB' and for all $pi \in \mathrm{Dom}(PP')$

$$cenv' \vdash PP'\ pi : t'_{pi}\ \&\ b_{pi}$$

for some t'_{pi} and b_{pi} with $b_{pi} \sqsubseteq\!\!\approx PB'\ pi$. (It is possible to take $t'_{pi} = t_{pi}$ whenever t_{pi} is defined.) □

The proof of Theorem 6 is by induction on the rules for concurrent evaluation. The soundness of the ordering defined in Table 3 is important for the proof of Theorem 6. However, it may be interesting to note that the laws **P1**, **P2**, **C1**, **C2**, **S1**, **E1**, **E2** and **J1** suffice.

5 Analysis of Finite Communication Topology

We shall say that e has a *finite communication topology* if there exists n and m such that whenever

$$\emptyset\ \&\ [pi_0 \mapsto e] \longrightarrow^{ev_1}_{ps_1} \ldots \longrightarrow^{ev_k}_{ps_k} CI\ \&\ PP$$

then

$$\#\mathrm{CHAN}\{ev_1, \ldots, ev_k\} \leq n \text{ and}$$
$$\#\mathrm{FORK}\{ev_1, \ldots, ev_k\} \leq m$$

where $\#\mathrm{CHAN}(X)$ is the cardinality of $\{\textsc{chan}\ ci \mid \textsc{chan}\ ci \in X\}$ and $\#\mathrm{FORK}(X)$ is the cardinality of $\{\textsc{fork}\ pi \mid \textsc{fork}\ pi \in X\}$. Thus the execution of e will cause at most n channels and m processes to be created.

Similarly, we say that b has a *finite communication topology* if there exists n and m such that whenever

$$[pi_0 \mapsto b] \Longrightarrow^{a_1}_{ps_1} \ldots \Longrightarrow^{a_k}_{ps_k} PB$$

then

$$\#\mathrm{CHAN}\{a_1, \ldots, a_k\} \leq n \text{ and}$$
$$\#\mathrm{FORK}\{a_1, \ldots, a_k\} \leq m$$

where $\#\mathrm{CHAN}(X)$ is the cardinality of $\{t\ \textsc{chan}\ r \mid t\ \textsc{chan}\ r \in X\}$ and $\#\mathrm{FORK}(X)$ is the cardinality of $\{\textsc{fork}\ b \mid \textsc{fork}\ b \in X\}$.

Example 7 Consider the following behaviours

(i) $\textsc{rec}\ \beta.\ (t\ \textsc{chan}\ r + (r!t; \beta))$

(ii) $\textsc{rec}\ \beta.\ (r?t + (t\ \textsc{chan}\ r; \beta))$

(iii) $\textsc{rec}\ \beta.\ (t\ \textsc{chan}\ r + (r!t; \beta; \beta))$

(iv) $\textsc{rec}\ \beta.\ (\epsilon + (r!t; \beta; \beta))$

(v) $\textsc{rec}\ \beta.\ (t\ \textsc{chan}\ r + \textsc{fork}(r?t; \beta))$

$$\vdash \epsilon : (0,0)\ \&\ \emptyset, \emptyset$$

$$\vdash r!t : (0,0)\ \&\ \emptyset, \emptyset$$

$$\vdash r?t : (0,0)\ \&\ \emptyset, \emptyset$$

$$\vdash t\ \textsc{chan}\ r : (1,0)\ \&\ \emptyset, \emptyset$$

$$\frac{\vdash b : (n,m)\ \&\ \emptyset, \emptyset}{\vdash \textsc{fork}\ b : (n, m+1)\ \&\ \emptyset, \emptyset}$$

$$\frac{\vdash b_1 : (n_1, m_1)\ \&\ V_1, D_1 \quad \vdash b_2 : (n_2, m_2)\ \&\ V_2, D_2}{\vdash b_1; b_2 : (n_1 + n_2, m_1 + m_2)\ \&\ V_1 \cup V_2, D}$$

if $V_1 \neq \emptyset$ then $(n_2, m_2) = (0,0)$
if $V_2 \neq \emptyset$ then $(n_1, m_1) = (0,0)$
if $V_1 = \emptyset\ \vee\ V_2 = \emptyset$ then $D = D_1 \cup D_2$
else $D = V_1 \cup V_2$

$$\frac{\vdash b_1 : (n_1, m_1)\ \&\ V_1, D_1 \quad \vdash b_2 : (n_2, m_2) \& V_2, D_2}{\vdash b_1 + b_2 : (\max\{n_1, n_2\}, \max\{m_1, m_2\})\ \&\ V, D}$$

if $V = V_1 \cup V_2$ and $D = D_1 \cup D_2$

$$\frac{\vdash b : (n,m)\ \&\ V, D}{\vdash \textsc{rec}\ \beta.\ b : (n,m)\ \&\ V \setminus \{\beta\}, D \setminus \{\beta\}}$$

if $\beta \in D$ then $(n,m) = (0,0)$

$$\vdash \beta : (0,0)\ \&\ \{\beta\}, \emptyset$$

Table 10: Test for Finite Communication Topology

Here (i) has a finite topology: at most one channel is created. Example (ii) does not have a finite topology since any number of channels may be created. Although (iii) is only a slight variation of (i) it does not have a finite topology: any number of channels may be created. In (iv) we do have a finite topology and in (v) we do not since any number of processes may be created. □

To test whether a behaviour b has a finite communication topology we shall introduce a predicate

$$\vdash b : (n,m)\ \&\ V, D$$

which intuitively expresses that b has a finite communication topology where at most n channels and m processes are created. The predicate is defined in Table 10. The set V is the set of free behaviour variables of b and D is a subset of V consisting of those behaviour variables that are on a path in b with more than one behaviour variable. These so-called *dangerous variables* need special attention when handling recursion as explained below.

Let us first explain some of the simple cases. The clauses for ϵ, $r!t$, $r?t$ and $t\ \textsc{chan}\ r$ should be straightforward. For $\textsc{fork}\ b$ we have to ensure that b contains

no free behaviour variables — otherwise an encapsulation into a recursive construct (as in Example 7 (v)) will cause an unbounded number of processes to be created. For the choice-construct we simply take the maximum of the numbers of channels and processes created by each of the components.

In order for a behaviour of the form REC β. b to have a finite communication topology it must create the same number of channels and processes as its one-level unfolding $b[\beta \mapsto \text{REC } \beta.\ b]$. In the analysis we cannot perform this unfolding because we want the analysis to be *structural*. So instead we have to consider processes with free behaviour variables and here we shall boldly claim that β creates no channels and no processes. Clearly this would be correct if we require all recursive behaviours not to create any channels and processes but this is too restrictive. We shall therefore introduce the set D of dangerous variables to keep track of when it is really necessary that the behaviour of a recursive construct creates no channels and processes. Such situations arise in connection with the sequencing $b_1; b_2$. If b_1 contains a free behaviour variable then the channels and processes created by b_2 can be iterated any number of times by unfolding the encapsulating recursive binding of the behaviour variable. So we must ensure that b_2 does not create any channels and processes. This is complicated by the possibility that b_2 could contain free behaviour variables which (maybe wrongly) have been assumed to create no channels and processes. We must therefore ensure that these free behaviour variables never can generate any channels and processes; we do this by including them in the set of dangerous behaviour variables. We impose similar restrictions in the dual situation. In this way the behaviour t CHAN $r; \beta$ of (ii) in Example 7 will classify β as a dangerous variable because t CHAN r creates a channel, whereas $r!t; \beta$ of (i) will not classify β as dangerous. The behaviour $r!t; \beta; \beta$ of (iii) and (iv) causes no problems but β is recorded as a dangerous variable and when analysing the encapsulating REC construct, (iii) will be rejected because its body may create a channel whereas (iv) will be accepted.

The following result expresses the soundness of the analysis:

Theorem 8 If $\vdash b : (n, m)\ \&\ \emptyset, \emptyset$ and

$$[pi_0 \mapsto b] \Longrightarrow^{a_1}_{ps_1} \ldots \Longrightarrow^{a_k}_{ps_k} PB$$

then

$$\#\text{CHAN}\{a_1, \ldots, a_k\} \leq n \text{ and}$$
$$\#\text{FORK}\{a_1, \ldots, a_k\} \leq m. \qquad \square$$

To prove this result we shall first show that the property expressed by the predicate of Table 10 is preserved by sequential evolution of behaviours:

Proposition 9 Assume $\vdash b : (n, m)\ \&\ \emptyset, \emptyset$. If

$$b \Rightarrow^p \flat$$

then there exists n_0, m_0, n' and m' such that $\vdash p : (n_0, m_0)\ \&\ \emptyset, \emptyset$ and $\vdash \flat : (n', m')\ \&\ \emptyset, \emptyset$ and furthermore

$$n_0 + n' \leq n \text{ and}$$
$$m_0 + m' \leq m. \qquad \square$$

Here we have extended the predicate of Table 10 to configurations by taking

$$\vdash \surd : (0, 0)\ \&\ \emptyset, \emptyset$$

The proof of Proposition 9 is by induction on the inference of $b \Rightarrow^p \flat$. To handle recursion we rely on the following lemma:

Lemma 10 Assume $\vdash b : (n, m)\ \&\ V, D$ and $\vdash b_0 : (n_0, m_0)\ \&\ \emptyset, \emptyset$ where b_0 has no free behaviour variables and $D \neq \emptyset$ implies $(n_0, m_0) = (0, 0)$. Then

- if $\beta \in V \setminus D$ then
  $\vdash b[\beta \mapsto b_0] : (\max\{n, n_0\}, \max\{m, m_0\})\ \&\ V \setminus \{\beta\}, D$,
- if $\beta \in D$ then $\vdash b[\beta \mapsto b_0] : (n, m)\ \&\ V \setminus \{\beta\}, D'$
  where $D' \subseteq D \setminus \{\beta\}$, and
- if $\beta \notin V$ then $\vdash b[\beta \mapsto b_0] : (n, m)\ \&\ V, D$. $\square$

To prove the concurrent soundness of the analysis we define

$$\vdash PB : (n, m)$$

to mean that

$$\vdash \flat_1 : (n_1, m_1)\ \&\ \emptyset, \emptyset, \cdots, \vdash \flat_j : (n_j, m_j)\ \&\ \emptyset, \emptyset$$

and $n = n_1 + \cdots + n_j$ and $m = m_1 + \cdots + m_j$ where $PB = [pi_1 \mapsto \flat_1, \cdots, pi_j \mapsto \flat_j]$. We now have

Proposition 11 If $\vdash PB : (n, m)$ and

$$PB \Longrightarrow^{a_1}_{ps_1} \ldots \Longrightarrow^{a_k}_{ps_k} PB'$$

then there exists n' and m' such that $\vdash PB' : (n', m')$ and

$$n' + \#\text{CHAN}\{a_1, \ldots, a_k\} \leq n \text{ and}$$
$$m' + \#\text{FORK}\{a_1, \ldots, a_k\} \leq m. \qquad \square$$

Theorem 8 then directly follows from Proposition 11. Combining Theorem 6 and Theorem 8 we get

Corollary 12 If $\emptyset \vdash e : t \ \& \ b$ and $\vdash b : (n,m) \ \& \ \emptyset,\emptyset$ then e has a finite communication topology where at most n channels and m processes are created. □

Example 13 Returning to Example 2 recall that the latent behaviour of `process` is

$$b = \text{FORK } (\text{REC } \beta'.\ (\rho_1?\alpha_1;\ \beta;\ \rho_2!\alpha_2;\ \beta'))$$

and it is easy to verify that

$$\vdash b : (0,1) \ \& \ \emptyset,\emptyset$$

Together with Corollary 12 this shows that indeed only one process will be created when the function `process` is executed (provided that the argument does not create any processes).

The latent behaviour of the function `pipe` is

$$\begin{array}{lll} b' = \text{REC } \beta'. & (\text{FORK } (\text{REC } \beta''.\ (\rho?\alpha;\ \rho!\alpha;\ \beta'')) \\ + & \alpha \text{ CHAN } \rho; \\ & \text{FORK } (\text{REC } \beta''.\ (\rho?\alpha;\ \beta;\ \rho!\alpha;\ \beta''));\\ & \beta') \end{array}$$

and here we cannot deduce that $\vdash b' : (n,m) \ \& \ \emptyset,\emptyset$ because β' is included in the set of dangerous behaviour variables for the body of the REC-construct. However, if we specialise the program to lists of fixed length then we do get a finite communication topology. □

6 Conclusion

We have presented a process algebra for a polymorphic subset of CML and have proved its safety with respect to the semantics of CML. Using the process algebra we then developed a static analysis of the communication behaviour of a CML program and proved its correctness. The applicability of the analysis is obvious: knowing that only a finite number of channels and processes are to be created may facilitate that the computing resources may be used much more efficiently. Our approach is flexible: if, for example, only the number of processes are of interest then the analysis may be modified so as to allow an arbitrary number of channels. However, more research is needed on how to combine the analysis with a transformation that automatically rewrites a CML program into a form where the processes can be preloaded on a given architecture.

There is plenty of scope for variations on the analysis. Theorem 8 expresses the soundness of the analysis: if the analysis succeeds then the behaviour has a finite communication topology. One may ask whether the analysis is complete: if the behaviour has a finite communication topology will the analysis say so? This is not the case as the following example illustrates:

$$(\text{REC } \beta.\ \beta);(\text{REC } \beta.\ (t \text{ CHAN } r;\beta))$$

Here the second component causes the analysis to reject the behaviour. Nonetheless the behaviour has a finite communication topology: the looping of the first component prevents any channels from being created. Another example is

$$\text{FORK}(r?t);r?t;(\text{REC } \beta.\ (t \text{ CHAN } r;\beta))$$

Again the analysis will reject the behaviour because of its last component. However, after forking the first process the behaviour will dead-lock and thereby prevent any channels from being generated. To get a complete analysis (with respect to the evolution of behaviours) one may extend the analysis with some form of termination/dead-lock analysis. Alternatively, one may study completeness only for the class of behaviours that at any stage of their evolution has the possibility of terminating. We conjecture that the analysis is indeed complete with respect to this class. Obviously, we cannot hope for a complete analysis with respect to the semantics of CML programs.

The analysis does not take into account that some processes may terminate before others are started. Consider the following example

$$\text{REC } \beta.\ (\epsilon + \text{FORK}(r!t;\beta))$$

This behaviour will be rejected by the analysis because any number of processes may be forked. However, at any time essentially all but one process will have terminated so one might regard the behaviour as having a finite communication topology.

Acknowledgements

We would like to thank Uffe Engberg for many fruitful discussions. Also Torben Amtoft, Kim Guldstrand Larsen, Bent Thomsen and Mads Tofte provided useful comments. The research has been supported by the DART project funded by The Danish Research Councils.

References

[1] D. Berry, R. Milner, D.N. Turner: A semantics for ML concurrency primitives. Proceedings of POPL'92, ACM Press, 1992.

[2] A. Giacalone, P. Mishra, S. Prasad: Operational and Algebraic Semantics for Facile: A Symmetric Integration of Concurrent and Functional Programming. Proceedings of ICALP '90, SLNCS 443, 1990.

[3] J.M. Lucassen, D.K. Gifford: Polymorphic Effect Systems. Proceedings of POPL'88, ACM Press, 1988.

[4] R. Milner: *Communication and Concurrency.* Prentice Hall, 1989.

[5] R. Milner, M. Tofte, R. Harper: *The Definition of Standard ML.* MIT Press, 1990.

[6] F. Nielson: The Typed Lambda-Calculus with First-Class Processes. Proceedings of PARLE'89, SLNCS 366, 1989.

[7] F. Nielson, H.R. Nielson: From CML to Process Algebras. Proceedings of CONCUR'93, SLNCS 715, 1993.

[8] J.H. Reppy: CML: A Higher-order Concurrent Language. Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, ACM Press, 1991.

[9] J.H. Reppy: Higher-Order Concurrency. Ph.D.-Thesis, Report 92-1285, Department of Computer Science, Cornell University, 1992.

[10] J.-P. Talpin, P. Jouvelot: The Type and Effect Discipline. Proceedings of LICS'92, 1992.

[11] B. Thomsen: A Calculus of Higher Order Communicating Systems. Proceedings of POPL'89, ACM Press, 1989.

[12] B. Thomsen: Polymorphic sorts and types for concurrent functional programs. Technical report ECRC-93-10, 1993.