Towards a Semantic Theory of CML *

W. Ferreira
M. Hennessy
University of Sussex

February 21, 1995

Abstract

A simple untyped language based on *CML*, Concurrent ML, is defined and analysed. The language contains a *spawn* operator for initiating new independent threads of computation and constructs for the exchange of data between these threads. A denotational model for the language is presented where denotations correspond to computations of values rather than simply values. It is shown to be fully abstract with respect to a behavioural preorder based on contextual testing.

1 Introduction

The language Concurrent ML (*CML*), [18], is one of a number of recent languages which seeks to combine aspects of functional and concurrent programming. Standard ML, [19], is augmented with the ability to spawn off new independent threads of computation. Further constructs are added to enable these threads to synchronise and exchange data on communication channels. As it includes higher-order objects, which can be exchanged between threads as data, new channel name generation, and the ability to form abstractions over communication behaviours using the concept of `event` types, *CML* is a sophisticated language. Although it has been implemented there has been very little work on its semantic foundations.

There have been a number of attempts at giving an operational semantics, usually in terms of a reduction relation, to core subsets of the language. For example in [18, 2] the core language λ_{cv} is given a two-level operational semantics which results in a reduction relation between multi-sets of language expressions. We aim to extend this type of work in order to build more abstract semantic theories, encompassing both behavioural equivalences and denotational models.

As a first step in this direction we consider in this paper a relatively simple language which nevertheless contains some of the key features of *CML*. It is a language for the evaluation of simple untyped expressions based on the standard construction $let\ x = e_1\ in\ e_2$ to which is added a *spawn* operator for introducing new threads of computation. To enable these threads to cooperate a range of constructs, based on those of *CCS*, for receiving and sending values is also added. The resulting language is more powerful than the *fork calculus*, [7], and the language considered in [1] as computation threads have the ability to exchange data. It is also more powerful than the value-passing process algebra of [9] as not only can expressions exchange values as data but the evaluation of expressions can terminate in the production of values. More importantly in [9] the calculation of values is computationally trivial and does not affect the communication behaviour of expressions whereas with our present language both these are mutually dependent.

In Section 3 we give the syntax of our language and an operational semantics. This is more general than the corresponding reduction relations of [18, 2], as it also determines the communication potentials of expressions and their ability to produce values; the operational semantics is given in terms of an extended *labelled transition system*. This would enable us to define a notion of bisimulation equivalence for expressions but instead in this paper we consider a Morris style observational preorder, [15], based on the ability to guarantee the production of values.

Properties of the operational semantics are investigated in Section 4. These are encapsulated in the definition of *value production systems*, extensions of *labelled transition systems* by new actions representing the production of values. These new actions are incorporated into the operational semantics

*This work has been supported by the ESPRIT/BRA CONCUR2 project and the EPSRC grant GR/H16537

in such a way that the natural *monadic laws* suggested in [14] for the *let ... in ...* construct are satisfied and furthermore the *spawn* operator can be explained in terms of a parallel construct.

A behavioural preorder based on a natural notion of observations is defined in Section 5. Expressions in the language are still designed to evaluate to, or produce values. So the basic observation of an expression is that it *guarantees* the production of a value and we then define $e_1 \sqsubseteq e_2$ if every observation which can be made of e_1 can also be made of e_2. The remainder of the paper is devoted to building a fully-abstract denotational model for this preorder. In Section 6 we first outline the general structure which any reasonable denotational model should have, which we call a Natural interpretation. There are two independent sources for requirements. The first, viewing expressions as representing processes, simply suggests a domain of processes on which the standard processes constructors can be interpreted. The second, viewing expressions as representing *computations* of values, suggests a monadic structure as in [14]. The requirements resulting from the latter view are expressed in terms of a degenerate form of Kliesli triples.

We then proceed, in Section 7, to construct a particular Natural Interpretation. The starting point is the value passing version of Acceptance Trees, [8], considered in [9] which is extended to a new model **D** to take into account the ability of expressions to produce values. However, as pointed out above, the key point is the recognition that elements of the model correspond not to values but to *computations* of values and in order to obtain a monadic interpretation, [14], it is necessary to consider a *retract* of **D**, called **E**. This is is shown to be fully-abstract with respect to $\sqsubseteq$ in Section 8 and we end with a brief comparison with related work.

2 Mathematical Preliminaries

In this section we review the mathematical constructions and notations used in the remainder of the paper. We recommend the reader to skip this section and to refer to it only when necessary.

We refer to algebraic cpos, [6], in which every non-empty directed set has a least upper bound as a *predomain*. If in addition it has a least element it is a *domain*; equivalently this means that every directed set has a least upper bound. If D is a predomain then $D_\perp$ is the domain obtained by adjoining a least element and $\lambda d \in D.d_\perp$ denotes the obvious injection.

A function $f\colon D \longrightarrow E$ from the predomain D to the predomain E is *continuous* if it preserves lubs of directed sets. We use $[D \longrightarrow E]$ to denote the set of continuous functions. Ordered pointwise it is also a predomain and even a domain whenever E has a least element, i.e is a domain. If $f\colon [D^n \longrightarrow E]$, where D and E are predomains we use $up(f)\colon [(D_\perp)^n \longrightarrow E_\perp]$ to denote its obvious strict extension.

More generally for any set X and predomain D the set of functions from X to D, $(X \longrightarrow D)$, is also a predomain when ordered pointwise and a domain if D has a least element. For any continuous function $f\colon D^k \longrightarrow D$ let $f^X\colon (X \longrightarrow D)^k \longrightarrow (X \longrightarrow D)$ be defined by $f^X(\underline{g})x = f(g_1(v), \ldots, g_k(v))$.

We use $(X \rightharpoonup_f D)$ to denote the set of partial functions from X to D with a non-empty finite domain. This is ordered by

$$f \leq g \text{ if } f(x) \leq_D g(x) \text{ for every } x \in \mathit{domain}(g),$$

which makes it into a predomain. Generalising functions from D^k to D to functions from $(X \rightharpoonup_f D)^k$ to $(X \rightharpoonup_f D)$ is somewhat more complicated and requires an extra function as parameter; we only consider the case $k = 2$. For any $h\colon [D^2 \longrightarrow D]$ let $h^+\colon (X \rightharpoonup_f D)^2 \longrightarrow (X \rightharpoonup_f D)$ be defined by

$$h^+(f,g)x = \begin{cases} h(f(x), g(x)) & x \in \mathit{domain}(f) \cap \mathit{domain}(g) \\ f(x) & x \in \mathit{domain}(f) - \mathit{domain}(g) \\ g(x) & x \in \mathit{domain}(g) - \mathit{domain}(f). \end{cases}$$

This function h^+ is not necessarily continuous but we do have:

Lemma 2.1 *If $h(x,y) \leq x$ and $h(x,y) \leq y$ then*

- *h^+ is continuous*
- *$^+$ itself is continuous when confined to such functions.* □

We often denote $h^+(f,g)$ by $f +_h g$.

A convenient method for isolating sub-domains of a given domain is by the use of *retracts*.

Definition 2.2 A *domain retract* over a domain D is a a strict continuous function $r{:}[D \longrightarrow D]$ satisfying

1. $r \circ r = r$
2. $r(k)$ is compact for every compact $k \in D$.

□

If r is a domain retract let $kernel(r)$ denote the set of its fixpoints, $kernel(r) = \{\, d \in D \mid r(d) = d \,\}$. This also coincides with the image of r.

Proposition 2.3 *If r is a domain retract then $kernel(r)$ is a domain.* □

Finally we recall some notation on *acceptance sets* , [8]. If $\mathcal{A}$ is a non-empty finite collection of finite subsets of a set X it is called an acceptance set (over X) if it satisfies

1. $A, B \in \mathcal{A}$ implies $A \cup B \in \mathcal{A}$
2. $A, B \in \mathcal{A}$ and $A \subseteq C \subseteq B$ implies $C \in \mathcal{A}$.

We use $\mid \mathcal{A} \mid$ to denote the basis of the acceptance set $\mathcal{A}$, i.e. $\cup\{\, A \mid A \in \mathcal{A} \,\}$ and $\mathcal{A}(X)$ to denote the set of all acceptance sets over X; ordered by reverse subset inclusion it is a predomain. We use three binary operators over $\mathcal{A}(X)$. For $\mathcal{A}, \mathcal{B} \in \mathcal{A}(X)$ let

1. $\mathcal{A} \wedge \mathcal{B}$ be $c(\mathcal{A} \cup \mathcal{B})$ where $c(\mathcal{C})$ is the smallest acceptance set containing the non-empty collection of finite subsets $\mathcal{C}$
2. $\mathcal{A} \vee \mathcal{B}$ be the acceptance set defined by $\{\, A \cup B \mid A \in \mathcal{A},\ B \in \mathcal{B} \,\}$
3. $\mathcal{A}\surd\mathcal{B}$, where $\surd$ is a distinguished element of X, be $c(\mathcal{C} \cup \mathcal{B})$ where $\mathcal{C} = \{\, A \mid A \in \mathcal{A},\ \surd \notin A \,\} \cup \{\, (A - \{\surd\}) \cup B \mid A \in \mathcal{A},\ \surd \in A,\ B \in \mathcal{B} \,\}$.

These three operators are continuous over the predomain consisting of $\mathcal{A}(X)$ ordered by reverse set inclusion.

3 The Language and its Operational Semantics

In this paper we consider a very simplified version of *CML*. It is based on a sequential language for evaluating expressions over some datatype, such as the Natural Numbers, whose main syntactic construct is the construction $let\ x = e_1\ in\ e_2$ Thus all notions of types are ignored and higher-order constructs are not considered. Nevertheless our language will incorporate some of the non-trivial features of *CML* .

Parallelism can be introduced into a sequential language for evaluating expressions by adding a new operator called *spawn* which can initiate a new computational thread. An abstract syntax for such a language could be given by the following:

$$\begin{array}{lcl} e & ::= & d \mid op(\underline{d}) \mid let\ x = e\ in\ e \mid b \mapsto e, e \mid spawn(e) \\ d & ::= & v \mid x \end{array}$$

Here v ranges over a set of basic values *Val* which we assume contains a distinguished value *null*, x over a set of variables *Var* and op over a set of function or operator symbols *Op*. We also assume the existence of a set of boolean expressions *BExp* ranged over by b.

The intended meaning of these constructs should be apparent but note that for convenience we only allow expressions of the form $op(\underline{e})$ when each e_i has one of the simple forms x or v. The effect of $op(\underline{e})$ for more general e_i can be obtained using the expression $let\ x_1 = e_1\ in\ let\ x_2 = e_2\ in\ \ldots op(x_1, \ldots, x_k)$. Thus the *let* construct is used in the language to make more explicit the order in which sub-expressions are evaluated.

The language as it stands is very limited. Although multiple evaluation threads can be activated, in expressions such as $let\ x = spawn(e_1)\ in\ e_2$, independent threads can not co-operate or share information. So we add to the language untyped versions of the communication primitives of *CML*,

- $n?\lambda x.e$, input a value along the communication channel n and apply the function $\lambda x.e$ to it
- $n!v.e_2$, output the value v along the channel n and then evaluate e_2,

and an untyped choice operator $e_1 + e_2$, meaning carry out the evaluation associated with the expression e_1 or that associated with e_2.

In addition to these operators which have their direct counterparts in *CML* we add a parallel operator $e_1 \mid e_2$, meaning carry out the evaluation of e_1 and e_2 concurrently. Such an operator does not appear in the syntax of *CML* but it enables us to express directly in the syntax of the language the states which are generated as the evaluation of an expression proceeds.
The complete abstract syntax of our language is given by the following:

$$\begin{array}{lcl} e & ::= & d \mid op(\underline{d}) \mid let\ x = e\ in\ e \mid b \mapsto e, e \mid spawn(e) \\ & & \mid n?\lambda x.e \mid n!d.e \mid e + e \mid e \mid e \\ & & \mid local\ n\ in\ e\ end \mid \delta \mid e \oplus e \mid let\ rec\ P\ in\ e \mid P \\ d & := & v \mid x \end{array}$$

The constructs not explained are

- $local\ n\ in\ e\ end$ – meaning that n is a local channel name for the evaluation of e,
- δ – an evaluation which can no longer proceed,
- $e_1 \oplus e_2$ – an internal or spontaneous choice between the evaluation of e_1 and e_2
- $let\ rec\ P\ in\ e$ – recursive definitions using a set of predefined expression names $P \in PN$.

The two constructs $e_1 \oplus e_2$ and δ are not essential as they can be defined using the other operators but they have proved to be convenient in the development of process algebras,[10, 8]. On the other hand $local\ n\ in\ e\ end$ does have a counterpart in *CML* although in our language we only have a predefined set of channel names N, over which n ranges, as opposed to the channel name generation facility of *CML*. Finally the facility for recursive definitions is modelled on that used in process algebras as recursion in *CML* is achieved using functional types.

We use *PExp* to denote the set of expressions generated by this abstract syntax and *CPExp* to denote the set of closed expressions; the standard definitions of free and bound occurrences of variables and process names apply and an expression is closed if it contains no free occurrence of a variable or a process name.

We now consider an operational semantics for *CPExp*. For the sake of simplicity we ignore the evaluation of boolean expressions. That is we assume that for each closed boolean expression b there is a corresponding truth value $[\![b]\!]$ and more generally for any boolean expression b and mapping ρ from variables to values there is a boolean value $[\![b]\!]\rho$. We also assume that for each operator symbol $op \in Op$ we have an associated function $[\![op]\!]$ over the set of values *Val* of the appropriate arity. The operational semantics for *CML*, in papers such as [7, 18, 2], are given in terms of a reduction relation between multisets of closed expressions, but because we have introduced the parallel operator $\mid$ our reduction relation is expressed simply as a binary relation $\xrightarrow{\tau}$ over closed expressions; $e \xrightarrow{\tau} e'$ means that in one step the closed expression e can be reduced to e'. Also these papers use a two-level approach to the operational semantics, the lower-level expressing reductions of individual expressions and the upper-level using these lower-level relations to define reductions between multisets of expressions. Instead, as is common for process algebras, we use auxiliary relations $\xrightarrow{n?v}$ and $\xrightarrow{n!v}$ to define our reduction relation $\xrightarrow{\tau}$.

There is one further ingredient. A sequence of reductions should eventually lead to the production of a value which in normal sequential languages also means that the computation has terminated. We use a special relation $\xrightarrow{\surd}$ to indicate the final production of a value. Thus the rule

$$v \xrightarrow{\surd} v$$

may be used to capture the idea that the simple expression v terminates in the production of the value v while the two rules

(VT) $\quad v \xrightarrow{\surd v} \delta, \quad op(\underline{v}) \xrightarrow{\surd w} \delta$ where $[\![op]\!](\underline{v}) = w$

$$(PT) \quad \frac{e_2 \xrightarrow{\surd v} e_2'}{e_1 \mid e_2 \xrightarrow{\surd v} e_1 \mid e_2'}$$

$$(BT) \quad \frac{e \xrightarrow{\surd v} e',\ [\![b]\!] = true}{b \mapsto e \xrightarrow{\surd v} e'}$$

$$(LT) \quad \frac{e \xrightarrow{\surd v} e'}{local\ n\ in\ e\ end \xrightarrow{\surd v} local\ n\ in\ e'\ end}$$

Figure 1: Operational semantics: value production rules

$e_1 \xrightarrow{\tau} e_1'$ implies $let\ x = e_1\ in\ e_2 \xrightarrow{\tau} let\ x = e_1'\ in\ e_2$
$e_1 \xrightarrow{\surd} v$ implies $let\ x = e_1\ in\ e_2 \xrightarrow{\tau} e_2[v/x]$

could adequately describe the semantics of local declarations of variables.

However the correct handling of the *spawn* construct requires some care. This is best discussed in terms of a degenerate form of local declarations; let $e_1; e_2$ be a shorthand notation for $let\ x = e_1\ in\ e_2$ where x does not occur free in e_2. We could therefore derive from above the natural rules:

$e_1 \xrightarrow{\tau} e_1'$ implies $e_1; e_2 \xrightarrow{\tau} e_1'; e_2$
$e_1 \xrightarrow{\surd} v$ implies $e_1; e_2 \xrightarrow{\tau} e_2$.

Intuitively $spawn(e_1); e_2$ should proceed by creating a new processor to handle the evaluation of e_1 which could proceed at the same time as the evaluation of e_2. However this requires a reinterpretation of the sequential composition operator ; as in [1]; $e_1; e_2$ no longer means when the evaluation of e_1 is finished start with the evaluation of e_2. Instead we interpret $e_1; e_2$ as "start the evaluation of e_2 as soon as an initialisation signal has been received from e_1". This initialisation signal is of course a $\surd$-move and the above judgement can be inferred if we allow the inferences

$spawn(e) \xrightarrow{\surd} e$
$e_1 \xrightarrow{\surd} e_1'$ implies $e_1; e_2 \xrightarrow{\tau} e_1' \mid e_2$.

In the second rule e_2 is initiated and its evaluation runs in parallel with that of the continuation, e_1', of e_1.

This discussion indicates a potential conflict between the two uses of the predicate $\surd$, one to produce values and the other to produce continuations. However this conflict can be resolved if we revise $\surd$ so that it has the type

$$\xrightarrow{\surd} \subseteq CPExp \times (Val \times CPExp).$$

When applied to a term it produces both a value and a continuation, for which we use the notation $e \xrightarrow{\surd v} e'$. The revised rule for simple values now becomes

$$v \xrightarrow{\surd v} \delta$$

where δ is the "deadlocked evaluation " .

Local declarations are now interpreted as follows:

$e_1 \xrightarrow{\tau} e_1'$ implies $let\ x = e_1\ in\ e_2 \xrightarrow{\tau} let\ x = e_1'\ in\ e_2$
$e_1 \xrightarrow{\surd v} e_1'$ implies $let\ x = e_1\ in\ e_2 \xrightarrow{\tau} e_1 \mid e_2[v/x]$.

$$(LtI)\quad \frac{e_1 \xrightarrow{\surd v} e_1'}{let\ x = e_1\ in\ e_2 \xrightarrow{\tau} e_1' \mid e_2[v/x]}$$

$$(SI)\quad spawn(e) \xrightarrow{\tau} e' \mid null$$

$$(ECI)\quad \frac{e_1 \xrightarrow{\surd v} e_1'}{e_1 + e_2 \xrightarrow{\tau} e_1' \mid v} \qquad \frac{e_2 \xrightarrow{\surd v} e_2'}{e_1 + e_2 \xrightarrow{\tau} e_2' \mid v}$$

$$(Com)\quad \frac{e_1 \xrightarrow{n?v} e_1',\ e_2 \xrightarrow{n!v} e_2'}{e_1 \mid e_2 \xrightarrow{\tau} e_1' \mid e_2'} \qquad \frac{e_1 \xrightarrow{n!v} e_1',\ e_2 \xrightarrow{n?v} e_2'}{e_1 \mid e_2 \xrightarrow{\tau} e_1' \mid e_2'}$$

$$(IC)\quad e_1 \oplus e_2 \xrightarrow{\tau} e_1 \qquad e_1 \oplus e_2 \xrightarrow{\tau} e_2$$

$$(Rec)\quad let\ rec\ P\ in\ e \xrightarrow{\tau} e[let\ rec\ P\ in\ e/x]$$

$$(ECA1)\quad \frac{e_1 \xrightarrow{\tau} e_1'}{e_1 + e_2 \xrightarrow{\tau} e_1' + e_2} \qquad \frac{e_2 \xrightarrow{\tau} e_2'}{e_1 + e_2 \xrightarrow{\tau} e_1 + e_2'}$$

Figure 2: Operational semantics: main reduction rules

Using these rules one can check that the evaluation of $spawn(e_1); e_2$ can proceed by initiating a thread for the evaluation of e_1 and then at any time launch a new thread which evaluates e_2.

The defining rules for the operational semantics are given in Figures 1,2,3. The first contains the rules for the relations $\xrightarrow{\surd v}$ while the second contains the most important rules for the reduction relation $\xrightarrow{\tau}$. The final Figure contains the rules for the external actions $\xrightarrow{n?v}$, $\xrightarrow{n!v}$ and routine rules for the reduction relation $\xrightarrow{\tau}$. Here μ ranges over the set of actions Act_τ which denotes $Act \cup \{\tau\}$, where Act denotes the set of external actions $\{\, n?v \mid n \in N,\ v \in Val\} \cup \{\, n!v \mid n \in N,\ v \in Val\}$.

Most of these rules have either already been explained or are readily understood. However it is worth pointing out the asymmetry in the termination rule for parallel, (PT). In the expression $e \mid e'$ only e' can produce a value using a $\surd$ action. Of course e can evaluate independently and indirectly contribute to this production by communicating with e' using the rule (Com). Also the rule for external choice (ECI) might be unexpected. It implies, for example, that if e_1 can produce a value v with a continuation e_1' then the external choice $e_1 + e_2$ can evolve to a state where the value v is available to the environment while the evaluation of continuation e_1' proceeds. Both these rules are designed to reflect the evaluation of *CML* programs as explained in [18].

4 Value Production Systems

The operational semantics of *PExp* determines a labelled transition system with a number of special properties. These are encapsulated in our definition of a *value production system*. First in the present circumstances it is reasonable to define a *labelled-value-transition system* as a collection $\langle E, Val, Act_\tau, \longrightarrow, \xrightarrow{\surd} \rangle$ where

- E is a set of *(closed) expressions*
- Val is a set of values such that $Val \subseteq E$
- $\longrightarrow \subseteq E \times Act_\tau \times E$
- $\xrightarrow{\surd} \subseteq E \times Val \times E$.

$(ECA2)$ $\dfrac{e_1 \xrightarrow{a} e_1'}{e_1 + e_2 \xrightarrow{a} e_1'}$ $\qquad$ $\dfrac{e_2 \xrightarrow{a} e_2'}{e_1 + e_2 \xrightarrow{a} e_2'}$

(PA) $\dfrac{e_1 \xrightarrow{\mu} e_1'}{e_1 \mid e_2 \xrightarrow{\mu} e_1' \mid e_2}$ $\qquad$ $\dfrac{e_2 \xrightarrow{\mu} e_2'}{e_1 \mid e_2 \xrightarrow{\mu} e_1 \mid e_2'}$

$(BoolA)$ $\dfrac{e_1 \xrightarrow{\mu} e_1', [\![b]\!] = true}{b \mapsto e_1, e_2 \xrightarrow{\mu} e_1'}$ $\qquad$ $\dfrac{e_2 \xrightarrow{\mu} e_2', [\![b]\!] = false}{b \mapsto e_1, e_2 \xrightarrow{\mu} e_2'}$

(LcA) $\dfrac{e \xrightarrow{\mu} e', chan(\mu) \neq n}{local\ n\ in\ e\ end \xrightarrow{\mu} local\ n\ in\ e'\ end}$

(LtA) $\dfrac{e_1 \xrightarrow{\mu} e_1'}{let\ x = e_1\ in\ e_2 \xrightarrow{\mu} let\ x = e_1'\ in\ e_2}$

(SA) $\dfrac{e \xrightarrow{\mu} e'}{spawn(e) \xrightarrow{\mu} spawn(e')}$

(In) $n?x.e \xrightarrow{n?v} e[v/x]$ for every value v

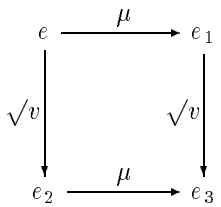
(Out) $n!v.e \xrightarrow{n!v} e$

Figure 3: Operational semantics: Auxiliary rules

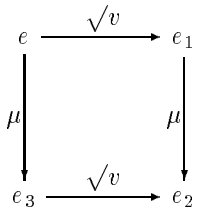
The operational semantics of *PExp* determines a labelled-value-transition system but it satisfies some additional properties.

Definition 4.1 A value production system, vps, is a collection system $\langle E, \delta, Val, Act_\tau, \longrightarrow, \xrightarrow{\surd} \rangle$ where

1. $\langle E, Val, Act_\tau, \longrightarrow, \xrightarrow{\surd} \rangle$ is a labelled-value-transition system
2. δ is a deadlocked expression, i.e. $\delta \not\xrightarrow{\mu}$ for every $\mu \in Act_\tau$ and $\delta \not\xrightarrow{\surd}$
3. the only move from the expression v is $v \xrightarrow{\surd v} \delta$, and $v \not\xrightarrow{\mu}$ for every $\mu \in Act_\tau$ and $v \xrightarrow{\surd w} e$ implies $e = \delta$ and $v = w$
4. *single-valuedness:* If $e \xrightarrow{\surd v} e'$ then $e' \xrightarrow{\surd w}$ for no w.
5. *value-determinacy:* $e \xrightarrow{\surd v} e'$ and $e \xrightarrow{\surd w} e''$ implies e' is e'' and $v = w$
6. *forward commutativity:* If $e \xrightarrow{\mu} e_1$ and $e \xrightarrow{\surd v} e_2$ then there exists an e_3 such that



7. *backward commutativity:* If $e \xrightarrow{\surd v} e_1$ and $e_1 \xrightarrow{\mu} e_2$ then there exists e_3 such that



□

Theorem 4.2 *The operational semantics of the previous section determines a vps with PExp as the set of expressions.*

Proof: The first three conditions are straightforward and the four others can be proved by rule induction on the operational semantics. As an example we outline the proof of *backward commutativity*. So suppose $e \xrightarrow{\surd v} e_1$ and $e_1 \xrightarrow{\mu} e_2$. The proof is by induction on the derivation of the transition $e \xrightarrow{\surd v} e_1$ and their are four cases, (VT), (PT), (BT) and (LT). As an example consider the second, (PT), when $e = f_1|f_2, e_1 = f_1|f_2'$ and $f_2 \xrightarrow{\surd v} f_2'$. There are three possibilities for the derivation $e_1 \xrightarrow{\mu} e_2$.

Case $f_2' \xrightarrow{\mu} f_2''$ **and** e_2 **is** $f_1 \mid f_2''$**:** Since $f_2 \xrightarrow{\surd v} f_2'$ then by induction there exists f_2''' such that $f_2 \xrightarrow{\mu} f_2''' \xrightarrow{\surd v} f_2''$ which implies $f_1|f_2 \xrightarrow{\mu} f_1|f_2''' \xrightarrow{\surd v} f_1|f_2''$.

Case $f_1 \xrightarrow{\mu} f_1'$ **and** e_2 **is** $f_1'|f_2'$**:** Then obviously $f_1|f_2 \xrightarrow{\mu} f_1'|f_2 \xrightarrow{\surd v} f_1'|f_2'$.

Case $f_1 \xrightarrow{a} f_1'$, $f_2' \xrightarrow{\overline{a}} f_2''$ **and** e_2 **is** $f_1' \mid f_2''$**:** This is a combination of the previous three cases.

□

We can now investigate properties of the operational semantics of *PExp* by deriving properties of an arbitrary vps. First we should point out that the *single-value* condition in the definition has considerable ramifications when considered in conjunction with the other conditions. Intuitively it says that if $e \xrightarrow{\surd v} e'$ then not only will e' be unable to produce any value but neither will any of its future derivatives; thus in any particular computation at most one value will ever be produced. Let P_E denote the set of *pure processes*,

$$P_E = \{\, e \in E \mid \forall s \in \{Act_\tau\}^*,\ e \stackrel{s}{\Longrightarrow} e' \text{ implies } e' \stackrel{\surd}{\not\longrightarrow} \,\}.$$

Lemma 4.3 *In any vps if* $e \xrightarrow{\surd v} e'$ *then* $e' \in P_E$.

Proof: Follows from *single-valuedness* and *backward commutativity*. □

However the main aim of this section is to show that our operational semantics endows the *let* construct with the intuitive properties one expects of it in the setting of a functional language and that the *spawn* operator can be modelled in a straightforward manner using the parallel construct. To this end note that the rules (PT), (Com) and (PA) can be used to define an operator | over any vps and in the sequel we only consider those vps which are closed with respect to this property, i.e. for every $e, e' \in E$ there is some expression $e \mid e' \in E$ whose behaviour is determined by these rules.

One consequence of the axioms defining a vps is that whenever $e \xrightarrow{\surd v} e'$ the behaviour of e is the same as $e' \mid v$, at least up to strong bisimulation. A suitable definition of strong bisimulation is as follows:

Definition 4.4 A symmetric relation $R \subseteq E \times E$ is called a *strong bisimulation* if it satisfies: $\langle e, e' \rangle \in R$ implies that

1. $e \xrightarrow{\mu} e_1$ implies $e' \xrightarrow{\mu} e'_1$ for some e'_1 such that $\langle e_1, e'_1 \rangle \in R$
2. $e \xrightarrow{\surd v} e_1$ implies $e' \xrightarrow{\surd v} e'_1$ for some e'_1 such that $\langle e_1, e'_1 \rangle \in R$.

Let $e \sim e'$ if $\langle e, e' \rangle \in R$ for some strong bisimulation R. □

Theorem 4.5 *In any vps if* $e \xrightarrow{\surd v} e'$ *then* $e \sim e' \mid v$.

Proof: Let

$$R = \{ \langle e, e' \mid v \rangle \mid e \xrightarrow{\surd v} e' \} \cup \{ \langle e, (e \mid \delta) \rangle \mid e \in P_E \}.$$

Using forward commutativity, value-determinacy and single-valuedness one can show that R is a strong bisimulation. □

This theorem demonstrates that values can only be produced by expressions in *PExp* in a very restricted manner. Essentially values can only be offered to the environment and subsequent behaviour can not depend on the value being absorbed by the environment. An immediate corollary of this is that the production of a value can not lead to an expression diverging; we say e *diverges*, written $e \Uparrow$ if there is an infinite sequence of derivations

$$e \xrightarrow{\tau} e_1 \xrightarrow{\tau} e_2 \xrightarrow{\tau} \ldots\ldots \xrightarrow{\tau} e_k \xrightarrow{\tau} \ldots\ldots$$

Corollary 4.6 *If* $e \xrightarrow{\surd v} e'$ *and* $e' \Uparrow$ *then* $e \Uparrow$. □

We now turn our attention to the properties of the *let* construct. At the abstract level of value production systems this is best studied by assuming there is a set of functions $\mathcal{F}$ from *Val* to E with the property that for each $e \in E$ there is an element $let\ x = e\ in\ f(x)$ whose actions are determined by the appropriate versions of the rules (LtI) and (LtA):

$$\frac{e \xrightarrow{\surd v} e'}{let\ x = e\ in\ f(x) \xrightarrow{\tau} e' \mid f(v)} \qquad \frac{e \xrightarrow{\mu} e'}{let\ x = e\ in\ f(x) \xrightarrow{\mu} let\ x = e'\ in\ f(x)}$$

In the case of the vps for the language *PExp* the set $\mathcal{F}$ consists of all functions $\lambda v \in Val.e[v/x]$ where e ranges over expressions in *PExp* which have at most x as a free variable.

One can easily show that in the abstract setting of a vps that the *let* construct satisfies properties such as

$$let\ x = e_1 \mid e_2\ in\ f(x) \sim e_1 \mid let\ x = e_2\ in\ f(x).$$

But unfortunately to obtain more interesting properties we have to work with respect to a slightly weaker equivalence than strong bisimulation.

Definition 4.7 A symmetric relation $R \subseteq E \times E$ is called a *mild bisimulation* if it satisfies: $\langle e, e' \rangle \in R$ implies that

1. for every $a \in Act$ $e \xrightarrow{a} e_1$ implies $\langle e_1, e'_1 \rangle \in R$ for some e'_1 such that $e' \xrightarrow{a} e'_1$ or $e' \xrightarrow{\tau} \xrightarrow{a} e'_1$
2. $e \xrightarrow{\surd v} e_1$ implies $\langle e_1, e'_1 \rangle \in R$ for some e'_1 such that either $e' \xrightarrow{\surd v} e'_1$ or $e' \xrightarrow{\tau} \xrightarrow{\surd v} e'_1$
3. $e \xrightarrow{\tau} e_1$ implies either $\langle e_1, e'_1 \rangle \in R$ for some e'_1 such that $e' \xrightarrow{\tau} e'_1$ or $e_1 \sim e'$.

Let $e \sim_m e'$ if $\langle e, e' \rangle \in R$ for some mild bisimulation R. □

Theorem 4.8 *In any vps*

1. $let\ x = e\ in\ Id(x) \sim_m e$ *where Id is the identity function*
2. $let\ x = v\ in\ f(x) \sim_m f(v)$ *for every value* v

3. $let\ x_2 = (let\ x_1 = e\ in\ f(x_1))\ in\ g(x_2) \sim let\ x_1 = e\ in\ (\lambda v.(let\ x_2 = f(v)\ in\ g(x_2)))(x_1)$.

Proof: In each case it is a matter of constructing a particular *mild-bisimulation* containing the required pairs.

For the first result we use

$$R_1 = \{\langle let\ x = e\ in\ Id(x), e\rangle\}\ \cup\ \{\langle e, e'\rangle \mid e \sim e'\}$$

and to prove it is a *mild-bisimulation* one needs to know that $e \sim e \mid \delta$ whenever $e \in P_E$.

For the second result the required *mild-bisimulation* is

$$R_2 = \{\langle let\ x = v\ in\ f(x)\ ,\ f(v)\rangle\}\ \cup\ Id,$$

while for the third we use

$$\begin{aligned} R_3 \quad = \quad & \{\langle let\ x_2 = (let\ x_1 = e\ in\ f(x_1))\ in\ g(x_2)\ ,\ let\ x_1 = e\ in\ (\lambda v.(let\ x_2 = f(v)\ in\ g(x_2)))(x_1)\rangle\}\ \cup \\ & \{\langle let\ x = e \mid e'\ in\ g(x)\ ,\ e \mid let\ x = e'\ in\ g(x)\rangle\} \end{aligned}$$

□

If we translate these results for the particular vps for $PExp$ it means that any behavioural equivalence contained in $\sim_m$ satisfies the following axioms:

$$\begin{aligned} let\ x = e\ in\ x &= e \\ let\ x = v\ in\ e &= e[v/x] \\ let\ x_2 = (let\ x_1 = e_1\ in\ e_2)\ in\ e_3 &= let\ x_1 = e_1\ in\ (let\ x_2 = e_2\ in\ e_3) \\ & \quad \text{provided } x_1 \notin fv(e_3). \end{aligned}$$

Note that Proposition 4.5 is crucial for the first law to hold and in the operational semantics this is ensured by the rules (ECI) and (SI). If in place of the former we had

$$\frac{e_1 \xrightarrow{\surd v} e_1'}{e_1 + e_2 \xrightarrow{\surd v} e_1'}$$

and its symmetric counterpart then the two expressions $v + n!v.\delta$ and $let\ x = v + n!v.\delta\ in\ x$ would not even be *weakly bisimular*, [13].

We end this section with a straightforward relationship between the *spawn* and parallel operators. Again let us assume that that for every expression e in a vps there is an expression $spawn(e)$ whose semantics is determined by the rule (SI) in Figure 2. The proof of the following result is left to the reader.

Proposition 4.9 *In any vps* $spawn(e) \sim_m e \mid null$ □

5 An Observational Preorder on Processes

We intend to construct a denotational model for the language but first, in this section, we present the behavioural preorder which we would wish the model to faithfully represent. It is based on the contextual preorder originally defined in [15] for the λ-calculus. The general method to construct a contextual preorder is to consider a set of observations appropriate to the language at hand and to say that the program fragment p is refined by q if in all contexts $C[\,]$ for which observations can be made of $C[p]$ and $C[q]$, every successful observation of $C[p]$ is also a successful observation of $C[q]$. We will restrict ourselves to the use of *finite* contexts, i.e. contexts which do not use the recursion operator of the language. This restriction is really only to make the proofs of our results manageable and we conjecture that they remain true if this restriction were lifted.

For our language a reasonable notion of observation is the production of values but in view of the inherent nondeterminism of the language there are are least two reasonable adaptions, based loosely on the *may* and *must* testing of [8]. We concentrate on the latter which informally can be viewed as being based on the ability of expressions to *guarantee* the production of values.

A computation of a closed expression e is any maximal sequence (i.e. it is finite and cannot be extended or it is infinite) of τ derivations from e. Let $Comp(e)$ be the set of computations of e. For any $c \in Comp(e)$ let c_i denote the ith component of c. Then for any $v \in Val$ we say that e *must* v if for all $c \in Comp(e)$ there exists some i such that $c_i \xrightarrow{\surd v}$.

Definition 5.1

For closed terms $e_1, e_2 \in CPExp$ let $e_1 \sqsubseteq e_2$ if for all contexts $C[], C[e_1]$ *must* w implies $C[e_2]$ *must* w where w is a new distinguished value. □

Although this preorder looks quite similar to the *must* testing of [8, 9] there is an important difference. In those papers a process term e is tested by running it in parallel with a testing process T. Thus the only testing contexts allowed were of the form $[\ ] \mid T$ which makes the analysis of the preorder more tractable. Here the testing contexts can be constructed using any of the operators of the language, although, for simplicity, the use of recursion in the contexts is not allowed. This change brings the preorder more in line with the original idea of contextual preorders, as suggested by Morris, [15] but the requirement that the value being guaranteed, w, being new is important. For example without this we would have

$$\Omega \mid v \not\sqsubseteq \Omega$$

just by taking the empty context. Since according to the above definition $\Omega \mid v$ *must* v and obviously Ω *must* v. However these two terms are identified in the model presented later. The present formulation leads to a more tractable semantic theory but we hope to examine natural variations in future work.

Contextual preorders are not very easy to work with and $\sqsubseteq$ is no exception. Accordingly we define an alternative characterisation which is more amenable to investigation. This alternative characterisation is quite similar to that used in [9] but there are important differences. Moreover the characterisation theorem is considerably more difficult to prove in view of the use of arbitrary contexts as tests. First some notation. Recall that Act_τ denotes the set of actions $Act \cup \{\tau\}$. This style of notation is extended by letting Act_{Val_τ} denote the set of actions Act together with τ and $\surd v$ for every value v, and Act_{Val} this set minus τ. Recall from Lemma 4.3 that in a given computation expressions can only produce at most one value. So the set of sequences of actions a process can perform is a subset of $S = Act^* \cup \{\, s_1 \surd v s_2 \mid v \in Val, s_1, s_2 \in Act^* \,\}$. To associate such sets with expressions we use the standard notation for transition systems: $\stackrel{\epsilon}{\Longrightarrow}$ denotes $\xrightarrow{\tau}{}^*$, for $a \in Act_{Val}$ $\stackrel{a}{\Longrightarrow}$ denotes the *weak* relation $\xrightarrow{\tau}{}^* \circ \xrightarrow{a} \circ \xrightarrow{\tau}{}^*$ and for any $s \in S$ $\stackrel{s}{\Longrightarrow}$ is its obvious extension.

For closed terms e let

Definition 5.2

$$\begin{aligned}
\mathcal{L}(e) &= \{\, s \mid e \stackrel{s}{\Longrightarrow} \,\} \text{ - the language of } e \\
\mathcal{S}(e) &= \{\, c? \mid e \stackrel{c?v}{\Longrightarrow} \text{ for some } v \,\} \cup \\
&\quad\ \{\, c! \mid e \stackrel{c!v}{\Longrightarrow} \text{ for some } v \,\} \cup \\
&\quad\ \{\, \surd \mid \text{if } e \stackrel{\surd v}{\Longrightarrow} \text{ for some } v \,\} \text{ - the successors of } e \\
\mathcal{SA}(e,s) &= \{\, \mathcal{S}(e') \mid e \stackrel{s}{\Longrightarrow} e' \not\xrightarrow{\tau} \,\} \text{ - the acceptances of } e \text{ after } s.
\end{aligned}$$

□

We also need to extend the notation on divergent computations. Recall that $e \Uparrow$ means e can diverge. We use $e \Downarrow$ to be the negation of this, namely e has no divergent computations, and this is generalised to sequences in the following way:

$$\begin{aligned}
e \Downarrow \varepsilon \quad &\text{if} \quad e \Downarrow \\
e \Downarrow \mu.s \quad &\text{if} \quad e \Downarrow \text{ and } e \stackrel{\mu}{\Longrightarrow} e' \text{ implies } e' \Downarrow s \text{ where } \mu \in Act_{Val}.
\end{aligned}$$

With this notation we are now ready to define the alternative characterisation.

Definition 5.3 For closed terms $e_1, e_2 \in CPExp$ let $e_1 \ll e_2$ if for every $s \in S$, $e_1 \Downarrow s$ implies

1. $e_2 \Downarrow s$
2. for all $B \in \mathcal{SA}(e_2, s)$ there exists $A \in \mathcal{SA}(e_1, s)$ such that $A \subseteq B$.

□

The remainder of this section is devoted to proving that for all closed expressions e_1, e_2

$$e_1 \sqsubseteq e_2 \text{ if and only if } e_1 \ll e_2. \tag{1}$$

The proof in one direction, $\sqsubseteq \subseteq \ll$, is similar in structure to the corresponding proof in [9], Proposition 2.8; three classes of contexts are defined which, in turn, can be used to test for, and indeed characterise the relevant properties underlying the definition of $\ll$. Moreover the three classes are quite similar to those used in [9] but the proofs that they do indeed characterise the properties is considerably more subtle.

Let $C_{con}[.](s)$ be defined by

$$C_{con}[.](s) \;=\; let\ x = [.]\ in\ b!x.\delta \;\mid\; con(s) \text{ for fresh } b, x$$

where $cons(s)$ is defined by

$$\begin{array}{rcl} con(\varepsilon) & = & \delta \;+\; w \\ con(a?v.s) & = & a!v.con(s) \;+\; w \\ con(a!v.s) & = & a?x.(x = v \mapsto con(s), w) \;+\; w \\ con(\surd v.s) & = & b?x.(x = v \mapsto con(s), w) \;+\; w. \end{array}$$

Note that for any closed expression e if $C_{con}[e] \stackrel{\epsilon}{\Longrightarrow} e'$ then $e' \stackrel{\surd w}{\Longrightarrow}$, because of the construction of $C_{con}[\,]$. Therefore $C_{con}[e]$ *must* w if and only $C_{con}[e] \Downarrow$; thus the interest in next Proposition, in particular the third statement.

Proposition 5.4 *For every $e \in CPExp$*

1. *For $s \in Act^*, e \Downarrow s$ implies $e \mid con(s) \Downarrow$.*
2. *For $s \in S, e \stackrel{\surd v}{\Longrightarrow} e'$ and $e \Downarrow s$ implies $(e' \mid b!v.\delta) \mid con(s) \Downarrow$.*
3. *For $s \in S, e \Downarrow s$ implies $C_{con}[e](s) \Downarrow$.*

Proof:
In each case the proof is by induction on the length of s. But the proof of the second statement relies on the first and the third relies in turn on the second.

□

Proposition 5.5 *For every $e \in CPExp$ and $s \in S$, $e \Downarrow s$ if and only if $C_{con}[e](s)$ must w.*

Proof: One direction follows from the third part of the previous Lemma while the converse direction follows from

$$e \Uparrow s \text{ implies } C_{con}[e](s) \Uparrow$$

which is easily established using induction on s. □

Two more types of contexts are required. The first tests the inability of an expression to produce the sequence of actions $s\alpha$. where $\alpha \in Act_{Val}$ and $s \in S$. Let $C_{rej}[.](s, \alpha)$ denote

$C_{rej}[.](s,\alpha) \;=\; let\ x = [.]\ in\ b!x.\delta \mid rej(s,\alpha)$ for fresh b, x

where $rej(s,a)$ is defined as

$$\begin{array}{rcl} rej(\varepsilon, a?v) & = & a!v.\delta + w \\ rej(\varepsilon, a!v) & = & a?x.(x = v \mapsto \delta, w) + w \\ rej(\varepsilon, \surd v) & = & b?x.(x = v \mapsto \delta, w) + w \\ rej(a?v.s, \alpha) & = & a!v.rej(s,\alpha) + w \\ rej(a!v.s, \alpha) & = & a?x.(x = v \mapsto rej(s,\alpha), w) + w \\ rej(\surd v.s, \alpha) & = & b?x.(x = v \mapsto rej(s,\alpha), w) + w. \end{array}$$

The second tests for the presence of acceptance sets of a certain form. Let $R = \{a_1 \cdots a_k\}$ be a finite subset of $Act \cup \{\surd\}$ and s a sequence from S. Then define $C_{acc}[.](s,R)$ by

$C_{acc}[.](s,R) \;=\; let\ x = [.]\ in\ b!x.\delta \mid acc(s,R)$ for fresh b, x

where $acc(s,R)$ is defined as

$$\begin{array}{rcl} acc(\varepsilon, R) & = & acc(R) \\ acc(a?v.s, R) & = & a!v.acc(s,R) + w \\ acc(a!v.s, R) & = & a?x.(x = v \mapsto acc(s,R), w) + w \\ acc(\surd v.s, R) & = & b?x.(x = v \mapsto acc(s,R), w) + w \end{array}$$

and where $acc(R)$ is defined by

$$\begin{array}{rcl} acc(R) & = & \sum\{\, acc(a) \mid a \in R \,\} \\ acc(a?) & = & a!w.w \\ acc(a!) & = & a?x.w \\ acc(\surd) & = & b?x.w. \end{array}$$

Lemma 5.6

1. *If $e \Downarrow s\alpha$ then $s\alpha \notin \mathcal{L}(e)$ if and only if $C_{rej}[e](s,\alpha)$ must w.*
2. *If $e \Downarrow s$ then $acc(s,R)$ must w if and only if $A \cap R \neq \emptyset$ for every $A \in \mathcal{SA}(e,s)$.*

Proof: Similar to the corresponding results in Proposition 2.8 in [9], although the proofs are somewhat more delicate. □

Proposition 5.7 *For $e_1, e_2 \in PExp$, $e_1 \sqsubseteq e_2$ implies $e_1 \ll e_2$.*

Proof: The proof strategy is the same as Proposition 2.8 in [9] but using the contexts $C_{con}[\,](\,), C_{rej}[\,](\,)$, and $C_{acc}[\,](\,)$. □

The converse is considerably more difficult to establish and let us see why this is so. Suppose $e \ll e'$ and $C[e]$ *must* w. In order to establish $C[e']$ *must* w it is necessary to prove that in every computation from $C[e']$,

$$C[e'] \xrightarrow{\tau} c_1 \xrightarrow{\tau} c_2 \xrightarrow{\tau} \ldots \xrightarrow{\tau} c_k \xrightarrow{\tau} \ldots$$

there is some c_i such that $c_i \xrightarrow{\surd w}$. In [9] the only possible form for $C[e]$ is $e \mid p$ for some term e and consequently each c_i also has the form $e_i \mid p_i$. This means that every computation from $e \mid p$ can be decomposed into two sub-derivations, one from e' and one from p. Using $e \ll e'$ a sufficiently similar sub-derivation can be obtained from e which can be recomposed with that from p to construct a computation from $e \mid p$. This, together with $e \mid p$ *must* w, is sufficient to guarantee some c_i which can perform the required w.

Here, because of the use of arbitrary contexts, computations can not be easily decomposed, into the contribution from e and the corresponding contribution from the context $C[\,]$ and subsequently recomposed. To do so we have to develop an operational semantics for contexts using the idea of *action transducers* as in [12] and prove appropriate decomposition and recomposition theorems for sequences of actions from $C[e]$ in terms of sequences of actions from e and sequence transductions from $C[\,]$. All of this, which accumulates in the proof of

Proposition 5.8 *For closed expressions $e \ll e'$ implies $e \sqsubseteq e'$,*

is relegated to Appendix A.

Combining these two results we have

Theorem 5.9 *For all closed expressions e, e', $e \ll e'$ if and only if $e \sqsubseteq e'$.* □

6 Natural Interpretations

In the section we outline the general requirements on a model in order to interpret the language.

If we consider the core language introduced first in Section 3 there are two essential features, one based on *spawn*, for introducing concurrency, and the other the *let* construct common to functional programming languages. We discuss in turn the requirements imposed by these features on the notion of a suitable semantic domain.

As can be seen from the operational semantics, and Proposition 4.9, we use the parallel operator | in order to explain the behaviour of this process, and the behaviour of the parallel operator in turn can be explained in terms of the other operators we have introduced into the language, such as the input and output prefixes and choice. In short the concurrent aspect of the language can be catered for in any model which supports an interpretation of these operators. For convenience let Σ denote this set of operators:

- the constant δ
- a unary symbol $n!v$, for each $n \in N$ and each $v \in Val$
- a unary symbol $local\, n\ in\ \_\ end$, for each $n \in N$
- the binary symbols $+$, $\oplus$, $|$.

To interpret recursive terms over this alphabet we use a Σ-domain which consists of a domain D together with a continuous function f_D over D for each operator symbol f. To interpret input prefixing we require, for each channel $n \in N$ a continuous function of type $[[Val \longrightarrow D] \longrightarrow D]$.

Thus far the requirements are very similar to those used in [9] but we also need to interpret the *let* construct. Expressions in *CPExp* produce, or evaluate to, values from *Val* but because of their concurrent nature many significantly different evaluations can lead to the same value. Thus it is appropriate to think of elements of the domain D as *computations* of values from *Val* and the *let* construct as a mechanism for manipulating computations rather than values. A general denotational theory of programming languages based on the idea that programs denote computations rather than values has been developed in [14] using monads and we follow this approach. Indeed Theorem 4.8 indicates that any reasonable semantic interpretation can be viewed, at least intuitively, in terms of monads. Here we rely on their presentation as Kliesli triples, although because our language has on one datatype we will only require a degenerate form of these, based on two functions:

1. a function η_D which associates with each value v a trivial computation $\eta(v)$ for producing this value
2. a continuous functional $^{*^D}$ which extends a function f from values to computations to a function f^{*^D} from computations to computations.

Definition 6.1 An Interpretation for the language *PExp* consists of a 4-tuple $\langle D, in_D, \eta_D, {}^{*^D} \rangle$ where

1. D is a Σ-domain
2. $in_D : Chan \longrightarrow (Val \longrightarrow D) \longrightarrow D$
3. $\eta_D : Val \longrightarrow D$
4. $^{*^D} : [(Val \longrightarrow D) \longrightarrow [D \longrightarrow D]]$.

□

Given such an interpretation a denotational semantics for the language can be given as a function

$$D[\![\ ]\!] : PExp \longrightarrow [Env_{Val} \longrightarrow [Env_D \longrightarrow D]],$$

where Env_{Val} denotes the set of Val environments, i.e. mappings from the set of variables Var to the set of values Val and Env_D is the set of D environments, mappings from the set of process names PN to the domain D. This is defined by structural induction on expressions:

i) $D[\![x]\!]\rho\sigma = \eta(\rho(x))$

ii) $D[\![v]\!]\rho\sigma = \eta(v)$

iii) $D[\![op(\underline{d})]\!]\rho\sigma = \eta([\![op]\!](\rho(\underline{d})))$

iv) $D[\![f(\underline{e})]\!]\rho\sigma = f_D(D[\![\underline{e}]\!]\rho\sigma)$ for each $f \in \Sigma$

v) $D[\![let\ rec\ P\ in\ e]\!]\rho\sigma = Y\lambda\alpha.D[\![e]\!]\rho\sigma[\alpha/P]$

vi) $D[\![b \mapsto e_1, e_2]\!]\rho\sigma = \begin{array}{ll} D[\![e_1]\!]\rho\sigma & \text{if } [\![be]\!]\rho\sigma = T \\ D[\![e_2]\!]\rho\sigma & \text{if } [\![be]\!]\rho\sigma = F \end{array}$

vii) $D[\![n?x.e]\!]\rho\sigma = in_D\ n\ \lambda v.D[\![e]\!]\rho[v/x]\sigma$

viiii) $D[\![let\ x = e_1\ in\ e_2]\!]\rho = (\lambda v.D[\![e_2]\!]\rho[v/x])^{*^D}[\![e_1]\!]\rho$

ix) $D[\![spawn(e)]\!]\rho = [\![e]\!]\rho\ |_D\ null$

where Y is the least-fixpoint operator for continuous functions over D.

One can check that this this semantic function satisfies the standard "substitution lemma":

Lemma 6.2 $D[\![e]\!]\rho[v/x] = D[\![e[v/x]]\!]\rho$. □

However there are some reasonable requirements on the interpretation of the *let* construct which are best expressed as properties of the functions η and $^{*^D}$; these are derived directly from the monad laws given in [14].

Definition 6.3 An Interpretation is *Natural* if

1. $(\eta_D)^{*^D} = id_D$
2. $f^{*^D} \circ \eta_D = f$ for every f: $Val \longrightarrow D$
3. $f^{*^D} \circ g^{*^D} = (f \circ g^{*^D})^{*^D}$ for every f, g: $Val \longrightarrow D$.

□

These properties ensure that the interpretation of the *let* construct has some expected properties:

Proposition 6.4 *If D is a Natural Interpretation then*

1. $D[\![let\ x = e\ in\ x]\!] = D[\![e]\!]$
2. $D[\![let\ x = v\ in\ e]\!] = D[\![e[v/x]]\!]$
3. $D[\![let\ x_2 = (let\ x_1 = e_1\ in\ e_2)\ in\ e_3]\!] = D[\![let\ x_1 = e_1\ in\ (let\ x_2 = e_2\ in\ e_3)]\!]$ *provided* $x_1 \notin fv(e_3)$.

Proof: Each of these is a direct consequence of the corresponding constraint on Natural Interpretations, given in the previous definition. As an example we outline the proof of the second property and for

convenience we abbreviate $D[\![\ ]\!]$ to $[\![\ ]\!]$.

$$\begin{aligned}
[\![let\ x = v\ in\ e]\!]\rho &= (\lambda v.[\![e]\!]\rho[v/x])^{*^D}[\![v]\!] \\
&= (\lambda v.[\![e]\!]\rho[v/x])^{*^D}\eta_D(v) \\
&= ((\lambda v.[\![e]\!]\rho[v/x])^{*^D} \circ \eta_D)(v) \\
&= (\lambda v.[\![e]\!]\rho[v/x])(v) \quad \text{from condition 2 in Definition 6.3} \\
&= [\![e[v/x]]\!]\rho \ \text{from the previous Lemma.}
\end{aligned}$$

□

The aim of this paper is to provide a Natural Interpretation which is fully-abstract with respect to the behavioural preorder $\sqsubseteq$, i.e. which satisfies

$$D[\![e_1]\!] \leq D[\![e_2]\!] \text{ if and only if } e_1 \sqsubseteq e_2$$

for all expressions e_1, e_2.

7 Acceptance Trees

Here we first review the version of Acceptance Trees, [8], used in [9] to model a value-passing process language. We then discuss how it might be modified so as to interpret *PExp*.

Acceptance trees are models of processes where the branches are labelled by the actions a process can perform and each node represents the set of possible states a process can reach after the sequence of actions labelling the path from the root of the tree to the node. The nondeterministic behaviour of processes is represented by *acceptance sets*, see Section 2, attached to the nodes. In order to accommodate value-passing these trees are modified so that the branches from a node do not lead directly to another node but rather to a function from values to trees which represent the functional behaviour of the process on reception or transmission of a value along a channel.

We now define a recursive domain equation whose solution, in the category of domains with embeddings [17], is a formal representation of these trees. If N is the set of communication channels let cN represent the set $\{ n?, n! \mid n \in N \}$; nodes in the trees will be represented by acceptance sets over cN. Let I be a domain representing the *sequel* of a process after performing an input, and O a corresponding domain for output. Then $H_P(N, I, O)$ denotes the set of pairs $\langle \mathcal{A}, f \rangle$ which satisfy:

- $\mathcal{A}$ is an acceptance set over cN, i.e. $\mathcal{A} \in \mathcal{A}(cN)$,
- f is a function from cN to the disjoint union of I and O, i.e. $f : cN \multimap_f I + O$, such that $domain(f) =\mid \mathcal{A} \mid$
- $f(\alpha) \in I$ whenever $\alpha = n?$ for some channel n
- $f(\alpha) \in O$ whenever $\alpha = n!$ for some channel n

Elements of $H_P(N, I, O)$ can be ordered by:

$\langle \mathcal{A}, f \rangle \leq \langle \mathcal{B}, g \rangle$ if $\mathcal{B} \subseteq \mathcal{A}$ and $f \leq g$, i.e. $f(\alpha) \leq g(\alpha)$ for every α in the domain of g.

If both I and O are predomains then one can check that so is $H_P(N, I, O)$. Moreover H_P can be used to induce a functor in the category of predomains with continuous functions as morphisms. So we can solve the domain equation

$$\begin{aligned}
\mathbf{P} &= H_P(N, \mathbf{I}, \mathbf{O})_\perp \\
\mathbf{I} &= Val \longrightarrow \mathbf{P} \\
\mathbf{O} &= Val \multimap_f \mathbf{P}.
\end{aligned}$$

in the category of domains with embeddings as morphisms, [17]. The domain representing the input sequels is the set of functions from values to processes. Operationally processes can only output a finite

number of different values on any given channel and therefore the set of finite non-empty functions from values to processes is used for output sequels. Note that $\mathbf{O}$ is a predomain rather than a domain.

In addition to the input and output of values expressions from *PExp* can produce values, i.e. perform $\surd$ actions. Therefore to model *PExp* we need to modify the functor H_P by adding an extra component to handle the production of values. Let T be a domain of sequels to $\surd$. Then $H_D(N, I, O, T)$ is the set of pairs $\langle \mathcal{A}, f \rangle$ where now $\mathcal{A}$ is an acceptance set over $cN \cup \{\surd\}$ and f is as above except that in addition it satisfies

- $f(\surd)$ is in T whenever it is defined.

We also need to decide on a suitable component of the domain equation for the sequel to $\surd$. The production of values is very similar to the output of values but there are differences. We know from Section 4 that an expression will produce at most one value. This means that when a value is produced the sequel will never again produce a value; instead the sequel will behave like a process, an element of the domain $\mathbf{P}$. Therefore we can use as a model the initial solution to the domain equation

$$\begin{aligned} \mathbf{D} &= H_D(N, \mathbf{I}, \mathbf{O}, \mathbf{T})_\perp \\ \mathbf{I} &= Val \longrightarrow \mathbf{D} \\ \mathbf{O} &= Val \rightharpoonup_f \mathbf{D} \\ \mathbf{T} &= Val \rightharpoonup_f \mathbf{P}. \end{aligned}$$

Intuitively the previous domain $\mathbf{P}$ can be seen as a sub-domain of $\mathbf{D}$, simply the subset of elements of $\mathbf{D}$ which contain no occurrence of $\surd$ in any acceptance set. This can be expressed formally in terms of a domain retract, see Section 2, over $\mathbf{P}$ which eliminates all occurrences of $\surd$. This domain retract is defined as the least fixpoint of a functional, $Y\lambda X.\mathcal{E}$ where for each $X{:}[\mathbf{D} \longrightarrow \mathbf{D}]$ the expression $\mathcal{E}$ represents a function in $[\mathbf{D} \longrightarrow \mathbf{D}]$. This expression $\mathcal{E}$ is in turn defined in terms of a "tree manipulation" functional $R{:}[[\mathbf{D} \longrightarrow \mathbf{D}] \longrightarrow [H_D \longrightarrow H_D]]$, where H_D is an abbreviation for $H_D(N, \mathbf{I}, \mathbf{O}, \mathbf{T})$, defined by

$$R\ X\ \langle \mathcal{A}, f \rangle = \langle \mathcal{A}\backslash\surd, X \circ f \rangle$$

where $\mathcal{A}\backslash\surd = \{\, A - \{\surd\} \mid A \in \mathcal{A} \,\}$. Here, and in the sequel, we are using the convention that an expression of the form $\langle \mathcal{B}, g \rangle$, where the domain of g may be a superset of $\mid \mathcal{B} \mid$, actually represents the element $\langle \mathcal{B}, g\lceil_{|\mathcal{B}|} \rangle$ of H_D. Thus R when applied to a tree eliminates any occurrences of $\surd$ from the acceptance set decorating the root and then applies the "recursion variable" X to the sequels. Now if *fold* and *unfold* are the isomorphisms associated with the domain equation for $\mathbf{D}$ then

$$\lambda X.\mathit{fold} \circ \mathit{up}(R\ X) \circ \mathit{unfold}$$

has the type $[\,[\mathbf{D} \longrightarrow \mathbf{D}] \longrightarrow [\mathbf{D} \longrightarrow \mathbf{D}]\,]$. So we can define $\mathit{elim}_\surd{:}\,\mathbf{D} \longrightarrow \mathbf{D}$ to be its least fixpoint, i.e.

$$\mathit{elim}_\surd = Y(\lambda X.\mathit{fold} \circ \mathit{up}(RX) \circ \mathit{unfold})$$

Theorem 7.1

1. *$\mathit{elim}_\surd$ is a domain retract*
2. *The kernel of $\mathit{elim}_\surd$, i.e. $\{\, d \in \mathbf{D} \mid \mathit{elim}_\surd(d) = d \,\}$ is a domain which is isomorphic to $\mathbf{P}$.* □

Proof:

1. It is easy to see that $\mathit{elim}_\surd$ satisfies the conditions of Definition 2.2.
2. The obvious injection mapping from P to $\mathit{elim}_\surd(D)$ gives the required isomorphism.

□

In subsequent developments we tend to identify $\mathbf{P}$ with its injection into $\mathbf{D}$.

Most of the operators in $PExp$ have been interpreted over the domain $\mathbf{P}$ and they are easily modified to $\mathbf{D}$. For example constant δ is interpreted as $fold\langle\{\emptyset\},\emptyset\rangle$ and the input operator

$$in_{\mathbf{D}}\colon Chan \longrightarrow [Val \longrightarrow \mathbf{D}] \longrightarrow \mathbf{D}$$

is defined by

$$in_{\mathbf{D}}\ n\ f = fold\ \langle\{\{n?\}\}, n? \mapsto f\rangle_{\perp}.$$

As another example we give the definition of $\oplus_{\mathbf{D}}$. We use the notation for partial functions explained in Section 2. Also when describing an element $\langle\mathcal{A}, f\rangle$ of H_D the function f can be decomposed into three components, some of which may be empty; $f_{?}$, the restriction of f to elements of $|\ \mathcal{A}\ |$ of the form $n?$, $f_{!}$, the restriction to elements of the form $n!$ and $f_{\surd}$, the effect of f on $\surd$ if it is defined. Consequently we can define f by giving its three components $(f_{?}, f_{!}, f_{\surd})$.

The function $\oplus_{\mathbf{D}}$ is also interpreted as a least fixpoint, $Y\lambda X.\mathcal{E}$ where this time $\lambda X.\mathcal{E}$ has the type $[\,[D^2 \longrightarrow D] \longrightarrow [D^2 \longrightarrow D]\,]$. This functional has much the same structure as that used to define the retract $elim_{\surd}$; it has the form $\lambda X.fold \circ up(I\ X) \circ unfold$ where I is a "tree manipulating function" of type $[\mathbf{D}^2 \longrightarrow \mathbf{D}] \longrightarrow [H_D^2 \longrightarrow H_D]$. It is defined by

$$IX(\langle\mathcal{A}, f\rangle, \langle\mathcal{B}, g\rangle) = \langle\mathcal{A} \wedge \mathcal{B}, (f_{?}\ +_X{}^{Val}\ g_{?}, f!\ +_{X^+}\ g_{!}, f_{\surd}\ +_{X^+}\ g_{\surd})\rangle.$$

Given two trees $\langle\mathcal{A}, f\rangle, \langle\mathcal{B}, g\rangle$ it constructs a new tree whose root is labelled by $\mathcal{A} \wedge \mathcal{B}$ and whose sequels are obtained by applying the binary "recursion variable" X in a systematic manner to the sequels of the pair of inputs. Thus we define $\oplus_{\mathbf{D}}$ to be $Y(\lambda X.fold \circ up(I\ X) \circ unfold)$.

The interpretation of the external choice operator $+_{\mathbf{D}}$ is simpler in that it uses $\oplus_{\mathbf{D}}$ to merge the sequels of actions but it combines the initial acceptance sets in a slightly different manner. Let $+_{\mathbf{D}}$ be $fold \circ up(E) \circ unfold$ where $E\colon H_D^2 \longrightarrow H_D$ is the "tree manipulation function given by:

$$E(\langle\mathcal{A}, f\rangle, \langle\mathcal{B}, g\rangle) = \langle\mathcal{A} \vee \mathcal{B}, (f_{?}\ +_{\oplus}{}^{Val}\ g_{?}, f!\ +_{\oplus^+}\ g_{!}, f_{\surd}\ +_{\oplus^+}\ g_{\surd})\rangle.$$

The interpretation of the remaining operators from Σ are similar in nature and are given in Appendix B.

It remains to discuss the interpretation of the *let* construct and as explained in the previous section two functions are required. The first is straightforward. Let $\eta_{\mathbf{D}}\colon Val \longrightarrow \mathbf{D}$ be defined by letting $\eta_{\mathbf{D}}(v)$ be the tree

$$fold\langle\{\{\surd\}\}, \surd \mapsto (v \mapsto \delta_{\mathbf{D}})\rangle_{\perp}.$$

However the definition of $\ ^{*^{\mathbf{D}}}$ is considerably more complicated and we give it in two stages.

Let $H_{\surd}$ denote the predomain consisting of all elements $\langle\mathcal{A}, f\rangle$ of H_D such that $\surd \in |\ \mathcal{A}\ |$. Intuitively we wish to define an operation tr which takes a function $k\colon H_{\surd} \longrightarrow D$ and a "tree" $d \in D$ and modifies it by nondeterministically factoring in $k(n)$ at each node n at which the action $\surd$ occurs. The extension functional $\ ^{*^{\mathbf{D}}}$ will be defined using a special instance of the function k. The nondeterministic factoring is carried out by another binary choice operator, a variation on $+_{\mathbf{D}}$. Let $\oslash_{\mathbf{D}}$ denote $fold \circ up(S) \circ fold$ where $S\colon H_D^2 \longrightarrow H_D$ is a minor variation of E above, given by

$$S(\langle\mathcal{A}, f\rangle, \langle\mathcal{B}, g\rangle) = \langle\mathcal{A}\surd\mathcal{B}, (f_{?}\ +_{\oplus}{}^{Val}\ g_{?}, f!\ +_{\oplus^+}\ g_{!}, f_{\surd}\ +_{\oplus^+}\ g_{\surd})\rangle.$$

It will also be convenient to have a notation for conditionally applying a binary operator. If $\triangle$ is any infix binary operator and p is a predicate we use

$$e\ \triangle^p\ e'$$

to denote the expression $e \triangle e'$ if p evaluates to true and e if it evaluates to false.

For each $k\colon [H_{\surd} \longrightarrow \mathbf{D}]$ the function $tr\ k$ is defined as the least fixed point of a functional, $Y\lambda X.\mathcal{E}$ where again $\lambda X.\mathcal{E}$ is a functional of type $[\,[\mathbf{D} \longrightarrow \mathbf{D}] \longrightarrow [\mathbf{D} \longrightarrow \mathbf{D}]\,]$ and has the structure $\lambda X.fold \circ up(\,TR\ k\ X) \circ unfold$ where TR is a "tree manipulation" function of type

$$TR\colon [H_{\surd} \longrightarrow \mathbf{D}] \longrightarrow [\mathbf{D} \longrightarrow \mathbf{D}] \longrightarrow [H_D \longrightarrow H_D]$$

defined by

$$TR\ k\ X\langle\mathcal{A}, f\rangle = \langle\mathcal{A}, X \circ f\rangle_{\perp}\ \oslash^{\surd \in |\mathcal{A}|}\ k(\langle\mathcal{A}, f\rangle).$$

Thus when applied to a tree TR applies the recursion variable X to the sequels and if the acceptance set at the root contains an occurrence of $\surd$ it factors in an application of the function k. We leave the reader to check:

Lemma 7.2 *TR is continuous.* □

Therefore for any function k from $[H_{\surd} \longrightarrow \mathbf{D}]$

$$\lambda X.TR\ k\ X\colon [[D \longrightarrow D] \longrightarrow [H_D \longrightarrow H_D]]$$

and

$$fold \circ (\lambda X.up(TR\ k\ X)) \circ unfold\colon [[D \longrightarrow D] \longrightarrow [D \longrightarrow D]].$$

So we can define $tr\,k$ to be $Y(\lambda X.TR\ k\ X)$.

We now look at the application of tr to a particular class of functions generated by those in $(Val \longrightarrow \mathbf{D})$. For such an f let $f^v\colon H_{\surd} \longrightarrow \mathbf{D}$ be defined by

$$f^v\langle \mathcal{A}, g\rangle = \sum\nolimits_{\mathbf{D}} \{\, g_{\surd}(v) \mid_{\mathbf{D}} f(v) \;\mid\; v \in domain(g_{\surd}) \,\}.$$

where here $\sum\nolimits_{\mathbf{D}}$ represents the repeated application of $\oplus_{\mathbf{D}}$ to a finite non-empty set of elements of $\mathbf{D}$.

Definition 7.3 For any $f\colon Val \longrightarrow D$ let $f^{*^{\mathbf{D}}}$ denote $tr\,f^v$. □

We have now shown how to interpret each of the constructs from *PExp* in the domain $\mathbf{D}$ and therefore we have an Interpretation for *PExp*. Unfortunately it is not a Natural Interpretation as the requirement

$$\eta^{*^{\mathbf{D}}}_{\mathbf{D}} = id_{\mathbf{D}}$$

is not satisfied. The problem occurs because there are many compact elements in the domain $\mathbf{D}$ which are not denotable under this interpretation by expressions in *PExp*. A typical example is any d element of the form $\langle\{n!, \surd\}, f\rangle_\perp$ where $f(\surd) = \delta_{\mathbf{D}}$. One can check that $\eta^{*^{\mathbf{D}}}_{\mathbf{D}}\, d$ has the form $fold\,\langle \mathcal{A}, g\rangle_\perp$ where $\{\surd\} \in \mathcal{A}$ and therefore this must be different from d. However we can use a domain retract to cut down the model $\mathbf{D}$ so as to get a Natural Interpretation.

We have seen in Section 4 that the operational behaviour of expressions is constrained in that the properties of Value Production Systems are satisfied. To define a Natural Interpretation we need to isolate a subdomain of $\mathbf{D}$ which satisfies the semantic counterparts to these properties. To do so we use the function $\eta^{*^{\mathbf{D}}}_{\mathbf{D}}$.

Proposition 7.4

1. $\eta^{*^{\mathbf{D}}}_{\mathbf{D}}(d \oplus_{\mathbf{D}} d') = \eta^{*^{\mathbf{D}}}_{\mathbf{D}}(d) \oplus_{\mathbf{D}} \eta^{*^{\mathbf{D}}}_{\mathbf{D}}(d')$
2. $\eta^{*^{\mathbf{D}}}_{\mathbf{D}}$ *is a domain retract.* □

Proof: The proofs are straightforward but tedious and outlines may be found in [4]. □

Let $\mathbf{E}$ denote the kernel of $\eta^{*^{\mathbf{D}}}_{\mathbf{D}}$ which we know from Section 2 is a domain. This can be viewed as an Interpretation by using the functions already defined over $\mathbf{D}$. Specifically

1. for each symbol $f \in \Sigma$ let $f_{\mathbf{E}}$ be defined as $\eta^{*^{\mathbf{D}}}_{\mathbf{D}} \circ (f_{\mathbf{D}})\lceil_{\mathbf{E}}$,
2. the input function is defined as before, $in_{\mathbf{E}}\ n\ f = fold\,\langle\{\{n?\}\}, n? \mapsto f\rangle_\perp$
3. $\eta_{\mathbf{E}} = \eta_{\mathbf{D}}\lceil_{\mathbf{E}}$
4. for $f \in [Val \longrightarrow \mathbf{E}]$ let $f^{*^{\mathbf{E}}} = (f^{*^{\mathbf{D}}})\lceil_{\mathbf{E}}$.

With these definitions we have:

Proposition 7.5 $\mathbf{E}$ *is a Natural Interpretation.* □

The main result of the paper is:

Theorem 7.6 *The Natural Interpretation based on* $\mathbf{E}$ *is* fully-abstract, *i.e. for all expressions* $e,\ e' \in CPExp,\ \mathbf{E}[\![e]\!] \leq \mathbf{E}[\![e']\!]$ *if and only if* $e \sqsubseteq e'$.

The next section of the paper is devoted entirely to the proof of this theorem.

8 Relating Behavioural and Denotational Interpretations

In this section we outline the proof of full-abstraction:

For all closed expressions $e_1,\ e_2,\ \ e_1 \sqsubseteq e_2$ if and only if $\mathbf{E}[\![e_1]\!] \leq \mathbf{E}[\![e_2]\!]$.

We have already shown in Section 5 how $\sqsubseteq$ can be represented by the alternative characterisation $\ll$. In fact we can reformulate the ordering on elements of $\mathbf{D}$ in much the same way. This new ordering, also denoted by $\ll$, is *internally fully-abstract* with respect to $\leq$ on $\mathbf{D}$.

The definition of $\ll$ is a slight modification of the definition given in [9] for VPL. For each $\alpha \in Act_{Val}$ we define an infix partial function $\stackrel{\alpha}{\longrightarrow}$ by

$$\begin{array}{lll} T \stackrel{\alpha}{\longrightarrow} T' & \text{if} & \text{i) } \alpha \text{ is } a!v, unfold(T) = \langle \mathcal{A}, f\rangle \text{ and } T' \text{ is } f(c!)(v) \\ & \text{or} & \text{ii) } \alpha \text{ is } a?v, unfold(T) = \langle \mathcal{A}, f\rangle \text{ and } T' \text{ is } f(c?)(v) \\ & \text{or} & \text{iii) } \alpha \text{ is } \surd v, unfold(T) = \langle \mathcal{A}, f\rangle \text{ and } T' \text{ is } f(\surd)(v) \end{array}$$

Secondly we can define $\mathcal{A}(T, s)$ the acceptance sets of T after s as

1. $\mathcal{A}(T, \varepsilon) = \begin{cases} \mathcal{A} & \text{if } unfold(T) = \langle \mathcal{A}, f\rangle \\ \emptyset & \text{otherwise.} \end{cases}$
2. $\mathcal{A}(T, \alpha s) = \begin{cases} \mathcal{A}(T', s) & \text{if } T \stackrel{\alpha}{\longrightarrow} T' \\ \emptyset & \text{otherwise} \end{cases}$

Finally let $\Downarrow s$ for $s \in S$ be the least relation on trees satisfying the following rules.

1. $T \Downarrow \varepsilon$ if $T \neq \bot$
2. $T \Downarrow \alpha s$ if $T \Downarrow \varepsilon$ and $T \stackrel{\alpha}{\longrightarrow} T'$ implies $T' \Downarrow s$.

With these constructs we are ready to define the alternative characterisation.

Definition 8.1 For T, U, let $T \ll U$ if for every $s \in S, T \Downarrow s$ implies

1. $U \Downarrow s$
2. $\mathcal{A}(U, s) \subseteq \mathcal{A}(T, s)$.

□

Theorem 8.2 (Internal Full-Abstraction) *In* $\mathbf{D}$, $T \leq U$ *if and only if* $T \ll U$.

Proof: We refer the reader to the proof of Theorem 3.5.3 in [11], page 107, which is virtually identical. □

Recall that $\mathbf{E}$ is sub-domain of $\mathbf{D}$ and therefore it is sufficient to show that for closed terms

$$e_1 \ \ll \ e_2 \text{ if and only if } \mathbf{E}[\![e_1]\!] \ \ll \ \mathbf{E}[\![e_2]\!]. \tag{2}$$

To establish this it is sufficient to prove the two statements:

$$e \Downarrow s \text{ if and only if } [\![e]\!] \Downarrow s. \tag{3}$$

and

$$e \Downarrow s \text{ implies } \mathcal{A}([\![e]\!], s) = c(\mathcal{A}(e, s)). \tag{4}$$

The proof of these two require the use of *head normal forms*, or *hnfs*, and it is here that the proof diverges from that of the corresponding result in [9]; here we use head normal forms which are considerably more

complex. So we first define the required notion of *hnf*, show that convergent terms can always be transformed in one, and then show how they can be used in the proof of the two statements above. Because of the complexity of *hnfs* we need to introduce some notation before they can be defined. For the remainder of this section we use *Pre* to denote the set of *prefixes*, i.e. objects of the form $c!d$ or $c?\lambda x$, where d is a data expression, and we use α, β, $\gamma, \ldots$ to act as typical prefixes. A *sum form* is then any closed expression of the form

$$\sum_I \{\alpha_i.e_i\}$$

for some finite subset I of prefixes; if I is the empty set then this sum denotes δ.

Definition 8.3 [Base Standard Forms] Suppose $\mathcal{A}$ is a non-empty set of subsets of cN and that for each $a \in |\mathcal{A}|$ there is an expression e_a satisfying

1. If $a = c?$ then e_a has the form $c?\lambda x.e'$
2. If $a = c!$ then e_a has the form $\sum\{ c!v.f(v) \mid v \in dom(f) \}$ where $f \in (Val \rightharpoonup_f CPExp)$.

Then

$$\bigoplus \{ e_A \mid A \in \mathcal{A} \}$$

is a *basic standard form* if each e_A is the sum form $\sum\{ e_a \mid a \in A \}$. Here $\bigoplus$ denotes the application of the operator $\oplus$ to a non-empty set of expressions. □

These more or less correspond to the head normal forms used in [9] but here we have, in addition, to take into account the ability of expressions to produce values.

Definition 8.4 [Value Standard Forms] If V is a finite non-empty set of values and for each $v \in V$, e_v is a basic standard form then

$$\bigoplus \{ e_v \mid v \in V \}$$

is a *value standard form*. □

These value standard forms are used as the "$\surd$-derivatives" in the following definition of *head normal forms*, which is a generalisation of Definition 8.3.

Definition 8.5 [Head Normal Forms] Suppose $\mathcal{A}$ is a non-empty set of subsets of $cN \cup \{\surd\}$ and that for each $a \in |\mathcal{A}|$ there is an expression e_a satisfying

1. if $a = c?$ then e_a has the form $c?\lambda x.e'$
2. if $a = c!$ then e_a has the form $\sum\{ c!v.f(v) \mid v \in dom(f) \}$ where $f \in (Val \rightharpoonup_f CPExp)$.
3. if $a = \surd$ then e_a is a value standard form.

Then

$$\bigoplus \{ e_A \mid A \in \mathcal{A} \}$$

is a *hnf* if each e_A is a sum form $\sum\{ e_a \mid a \in A \}$. □

If e is a hnf then its structure is determined by a non-empty set of subsets $\mathcal{A}(e)$ and with each a in $|\mathcal{A}(e)|$ we can associate the subterm e_a as given in the above definition. Also if $\surd \in |\mathcal{A}(e)|$ then $e_\surd$ has the form

$$\bigoplus \{ e_v \mid v \in V \}$$

for some non-empty set of values V, where each e_v is a basic standard form. Here we use $V(e)$ to denote this set of values V so that for each $v \in V(e)$ we have an associated basic standard form e_v. Notice that the parallel operator $|$ may appear in head normal forms. Indeed their presence is essential; an expression such as $c?\lambda x.e \mid v$ has no semantically equivalent term not involving $|$.

The proof system we use for transforming expressions is given in Figure 4 and the associated equations and axiom schemas are outlined in Figures 5, 6, 7. It should be pointed out that these do not necessarily represent the most useful equations for manipulating expressions; they are simply the most convenient for the present purpose. The proof system is designed to produce judgements of the form $e = e'$ or $e \sqsubseteq e'$

$$\frac{}{e \sqsubseteq e} \qquad \frac{e \sqsubseteq e', e' \sqsubseteq e''}{e \sqsubseteq e''}$$

$$\frac{e_i \sqsubseteq e_i'}{f(\underline{e}) \sqsubseteq f(\underline{e'})} \text{ for each } f \in \Sigma' \qquad \frac{}{c!v.e = c!v.e}$$

$$\frac{}{c?x.e = c?y.(e[y/x])}\ y \notin fv(e) \qquad \frac{e \sqsubseteq e'}{e\sigma \sqsubseteq e'\sigma} \text{ for every equation } e \sqsubseteq e'$$

$$\frac{}{let\ rec\ P\ in\ e = e[let\ rec\ P\ in\ e/x]}$$

$$\frac{[\![b]\!] = true}{b \mapsto e, e' = e} \qquad \frac{[\![b]\!] = false}{b \mapsto e, e' = e'}$$

Figure 4: Proof System.

where both e and e' are closed expressions and we use the notation $e =_A e'$, $e \sqsubseteq_A e'$ respectively when such judgements can be made. The equations in Figure 5 are all very standard from [8] while those in Figure 6 are additions required for this language. Also, as might be expected a quite complicated interleaving law is required. In fact there are two, the first coinciding with that from [9], while the second is required because of the presence of values in head normal forms.

Most of the equations in the proof system are valid for **D** also. However the second *Let* equation and the last two *Parallel* equations in Figure 6 are only true in **E** because **D** does not model the preemptive power of the $\surd$ action in the presence of $+$.

Lemma 8.6 *If e, e' are hnfs then each of the expressions $e \oplus e'$, $e + e'$ and local n in e end can be transformed in the proof system to hnfs.*

Proof: [Outline] The proof can be divided into four parts.

1. Using standard techniques from [8, 9], one can show that if e, e' are sum forms then the expressions $e \oplus e'$, $e + e'$ and *local n in e end* can all be transformed into hnfs.
2. A *value-sum form* is any term of the form $e \mid v$ where e is a sum form. Using the equations one can also show that if e, e' are value sum forms then $e \oplus e'$, $e + e'$ and *local n in e end* can be transformed into hnfs.
3. Using the first two parts one can now show that if e, e' are hnfs then $e \oplus e'$ can be transformed into a hnfs.
4. The result now follows for the remaining operators since every hnf can be transformed into a term of the form $\sum \{ e_i \mid i \in I \}$ for some non-empty set I, where each e_i is either a sum form or a value-sum form.

□

Proposition 8.7 *For every closed term e, if $e \Downarrow$ then there is a hnf $hnf(e)$ such that $e =_A hnf(e)$.*

Proof: [Outline] The proof follows the structure of the corresponding result in [9], Proposition 4.3. In particular we need to use induction over a complicated well-ordering on expressions, in order to be able to apply induction after an application of the interleaving laws and the final *let* law in Figure 6. □

Testing equations:

$$\begin{array}{rcll}
X \oplus (Y \oplus Z) & = & (X \oplus Y) \oplus Z & (\oplus 1) \\
X \oplus Y & = & Y \oplus X & (\oplus 2) \\
X \oplus X & = & X & (\oplus 3) \\
X \oplus Y & \sqsubseteq & X & (S) \\
 & & & \\
X + (Y + Z) & = & (X + Y) + Z & (+1) \\
X + Y & = & Y + X & (+2) \\
X + X & = & X & (+3) \\
X + \delta & = & X & (+4) \\
 & & & \\
\mathit{local}\ n\ \mathit{in}\ X + Y\ \mathit{end} & = & \mathit{local}\ n\ \mathit{in}\ X\ \mathit{end} + \mathit{local}\ n\ \mathit{in}\ Y\ \mathit{end} & (+5) \\
\mathit{local}\ n\ \mathit{in}\ \alpha.X\ \mathit{end} & = & \begin{cases} \alpha.\mathit{local}\ n\ \mathit{in}\ X\ \mathit{end}, & \text{if } n \text{ not in } \alpha \\ \delta, & \text{otherwise} \end{cases} & \\
 & & & \\
X \oplus Y & \sqsubseteq & X + Y & (+\oplus 1) \\
\alpha.X + \alpha.Y & = & \alpha.(X \oplus Y) \quad \text{where } \alpha \in \mathit{Pre} & (+\oplus 2) \\
c?\lambda x.X + c?\lambda x.Y & = & c?\lambda x.X \oplus c?\lambda x.Y & (+\oplus 3) \\
c!d.X + c!d'.Y & = & c!d.X \oplus c!d'.Y & (+\oplus 4) \\
X \oplus (Y + Z) & = & (X \oplus Y) + (X \oplus Z) & (+\oplus 6)
\end{array}$$

Structural equations:

$$\begin{array}{rcl}
\mathit{local}\ n\ \mathit{in}\ X \oplus Y\ \mathit{end} & = & \mathit{local}\ n\ \mathit{in}\ X\ \mathit{end} \oplus \mathit{local}\ n\ \mathit{in}\ Y\ \mathit{end} \\
\mathit{let}\ x = X \oplus Y\ \mathit{in}\ Z & = & \mathit{let}\ x = X\ \mathit{in}\ Z \oplus \mathit{let}\ x = Y\ \mathit{in}\ Z \\
(X \oplus Y) \mid Z & = & (X \mid Z) \oplus (Y \mid Z) \\
X \mid (Y \oplus Z) & = & (X \mid Z) \oplus (Y \mid Z) \\
X + (Y \oplus Z) & = & (X + Y) \oplus (X + Z)
\end{array}$$

Figure 5: Standard Equations

Many of the operators on **E** behave the same as their counterparts on **D** when restricted to elements of **E**. For example if e_1, e_2 are elements of **E** then $e_1 \oplus_{\mathbf{D}} e_2 = e_1 \oplus_{\mathbf{E}} e_2$. This is a simple consequence of the definition of $\oplus_{\mathbf{E}}$ and Proposition 7.4. The same is true for the input and output operators which have the same definition for both **D** and **E**. This property is not true for $+_{\mathbf{E}}$ or $|_{\mathbf{E}}$ because of the preemptive nature of value production in these contexts. However we do have the following results:

$$\mathbf{E}[\![\sum\{\, e_a \mid a \in A \,\}]\!] = \sum_{\mathbf{D}}\{\, \mathbf{E}[\![e_a]\!] \mid a \in A \,\} \text{ where } \surd \notin A$$

and

$$\mathbf{E}[\![e \mid v]\!] = \mathbf{E}[\![e]\!] \mid_{\mathbf{D}} \mathbf{E}[\![v]\!].$$

It is also not difficult to show:

Proposition 8.8 *Suppose e is a term in sum-form, and $e_\surd$ is a term in value-standard form, then*

$$\mathbf{E}[\![e + e_\surd]\!] = (\mathbf{E}[\![e]\!] +_{\mathbf{D}} \mathbf{E}[\![e_\surd]\!]) \oplus_{\mathbf{D}} \mathbf{E}[\![e_\surd]\!].$$

□

We are now ready to prove the two required results above, (3) and (4). As a first step towards the proof of the first we have:

Lemma 8.9 *For every closed expression e, $e \Downarrow$ if and only if $\mathbf{E}[\![e]\!] \Downarrow$.*

Let equations:
Suppose $X = \sum_{i \in I} \alpha_i.X_i$ and $x \notin fv(\alpha_i)$,

$$let\ x = X\ in\ Y \quad = \quad \sum_{i \in I} \alpha_i.(let\ x = X_i\ in\ Y)$$

Suppose $X = \sum_{i \in I} \alpha_i.X_i, x \notin fv(\alpha_i)$ and $Y = \sum_{j \in J} \beta_j.Y_j$ for $\alpha_i, \beta_j \in Pre$.

$$let\ x = X + (Y \mid v)\ in\ Z \quad = \quad (\sum_{i \in I} \alpha_i.(let\ x = X_i\ in\ Z) + (Y \mid Z[v/x])) \oplus (Y \mid Z[v/x])$$

Parallel equations:

$$\begin{array}{rcl} spawn(X) & = & X \mid null \\ X \mid (Y \mid Z) & = & (X \mid Y) \mid Z \\ v \mid X & = & X \end{array}$$

Suppose $X = \sum_{i \in I} \alpha_i.X_i$, $Y = \sum_{j \in J} \beta_j.Y_j$ and $Z = \sum_{k \in K} \gamma_k.Z_k$ for $\alpha_i, \beta_j, \gamma_k \in Pre$, then

$$\begin{array}{rcl} (X + (Y \mid v)) \mid Z & = & ((X + Y) \oplus Y) \mid Z \\ (X \mid v) + (Y \mid w) & = & (X \mid v) \oplus (Y \mid w) \end{array}$$

Figure 6: New Equations

Proof: First suppose $e \Downarrow$. Then e has a hnf $hnf(e)$ and since $\mathbf{E}$ is a model of the proof system we have $\mathbf{E}[\![hsf(e)]\!] = \mathbf{E}[\![e]\!]$ and therefore it suffices to show $\mathbf{E}[\![hnf(e)]\!] \neq \bot$. By the structure of the hnf's and the previous Proposition and remarks we can "push" $\mathbf{E}[\![\ \ ]\!]$ through $hnf(e)$ so that at the top-level at least, the operators are those on $\mathbf{D}$. That $\mathbf{E}[\![hsf(e)]\!] \neq \bot$ is now obvious from the definitions of $+_{\mathbf{D}}$, $\oplus_{\mathbf{D}}$, $|_{\mathbf{D}}$, etc..

Conversely suppose that $\mathbf{E}[\![e]\!] \Downarrow$. We only outline the proof that $e \Downarrow$ as it is very similar to the corresponding proof, in Lemma 4.5 of [9]. If $\mathbf{E}[\![e]\!] \Downarrow$ then there is a finite approximation e' of e such that $\mathbf{E}[\![e']\!] \Downarrow$. Recall from [9] that a finite approximation of an expression e can be obtained by unwinding instances of recursion and substituting Ω for any sub-term. Consequently it is straightforward to show that $e' \ll e$. So $e \Downarrow$ will follow if we can show $e' \Downarrow$.

In fact we can show that for any finite expression e', i.e. any expression not involving the recursion construct, that $\mathbf{D}[\![e']\!] \Downarrow$ implies $e' \Downarrow$. (Since $\eta^{*^{\mathbf{D}}}_{\mathbf{D}}$ is strict it will also follow that $\mathbf{E}[\![e']\!] \Downarrow$ implies $e' \Downarrow$.) We add to the proof system in Figure 4 the equations

$$\begin{array}{rcl} \Omega & \sqsubseteq & X \\ X + \Omega & \sqsubseteq & \Omega \\ local\ n\ in\ \Omega\ end & = & \Omega \\ (X + \Omega) \mid Y & = & \Omega \\ X \mid (Y + \Omega) & = & \Omega \\ let\ x = \Omega\ in\ Y & = & \Omega \\ let\ x = X\ in\ \Omega & = & \Omega \end{array}$$

and let us write $e_1 =_{A\Omega} e_2$ if $e_1 = e_2$ can be derived in this extended system. Then one can show, by structural induction, that if e' is any finite expression either $e' =_{A\Omega} \Omega$ or there is an expression in head normal form $hnf(e')$ such that $e' =_{A\Omega} hnf(e')$. Now if $e' \Downarrow$ the former can not be true since all the equations are satisfied by $\ll$. It therefore follows that $\mathbf{D}[\![e']\!] \neq \bot$ since all hnfs are interpreted as non-bottom elements in $\mathbf{D}$. □

Suppose $X = \sum_{i\in I} \alpha_i.X_i, Y = \sum_{j\in J} \beta_j.Y_j$ and $Z = \sum_{k\in K} \gamma_k.Z_k$ for $\alpha_i, \beta_j, \gamma_k \in Pre$.

$$X \mid Y \;=\; \begin{cases} ext(X,Y) & \text{if } comms(X,Y) = \emptyset \\ (ext(X,Y) + int(X,Y)) \oplus int(X,Y) & \text{otherwise} \end{cases}$$

$$X \mid (Z + (Y \mid w)) \;=\; (ext_v(X,Z,Y \mid w) + int_v(X,Z,Y \mid w)) \oplus int_v(X,Z,Y \mid w)$$

where

$$ext(X,Y) \;=\; \sum\{\, a_i.(X_i \mid Y) \mid i \in I \,\} + \sum\{\, b_j.(X \mid Y_j) \mid j \in J \,\}$$

$$\begin{array}{rcl} int(X,Y) & = & \sum\{\, X_i[v/x] \mid Y_j \mid a_i = c?x, b_j = c!v \,\} \\ & \oplus & \sum\{\, X_i \mid Y_j[v/x] \mid a_i = c!v, b_j = c?x \,\} \end{array}$$

$$\begin{array}{rcl} ext_v(X,Z,Y \mid w) & = & \sum\{\, a_i.(X_i \mid (Z + (Y \mid w))) \mid i \in I \,\} \\ & + & \sum\{\, b_j.(X \mid (Y_j \mid w)) \mid j \in J \,\} \\ & + & \sum\{\, c_k.(X \mid Z_k) \mid k \in K \,\} \end{array}$$

$$int_v(X,Z,Y \mid w) \;=\; \begin{cases} (X \mid Y) \mid w & \text{if } comms(X,Y) \cup comms(X,Z) = \emptyset \\ (X \mid Y) \mid w & \text{otherwise} \\ \quad \oplus \sum\{\, X_i[v/x] \mid (Y_j \mid w) \mid a_i = c?x, b_j = c!v \,\} & \\ \quad \oplus \sum\{\, X_i \mid (Y_j[v/x] \mid w) \mid a_i = c!v, b_j = c?x \,\} & \\ \quad \oplus \sum\{\, X_i[v/x] \mid Z_k \mid a_i = c?x, c_k = c!v \,\} & \\ \quad \oplus \sum\{\, X_i \mid Z_k[v/x] \mid a_i = c!v, c_k = c?x \,\} & \end{cases}$$

Figure 7: Interleaving Laws

An important consequence of the definition of head-normal forms in [11] is that they form an exact syntactic representation of elements of the model, at least at the top-level. A Corollary of this is that

$$hnf(e) \xrightarrow{a} q \text{ if and only if } [\![hnf(e)]\!] \xrightarrow{a} [\![q]\!].$$

The head-normal forms used in our setting do not enjoy this property, but but we do have a weaker correspondence:

Lemma 8.10 *For every $a \in Act_{Val}$ and head normal form e*

1. *If $e \xrightarrow{\tau}{}^{*} \xrightarrow{a}$ if and only if $\mathbf{E}[\![e]\!] \xrightarrow{a}$;*
2. *$\mathbf{E}[\![e]\!] \xrightarrow{a} T$ implies $T = \mathbf{E}[\![\sum\{\, e'' \mid e \xrightarrow{\tau}{}^{*} \xrightarrow{a} e'' \,\}]\!]$.*

Proof: [Outline] The proof of both properties follows by examining the structure of $\mathbf{D}[\![e]\!]$ when e is a hnf and then using Proposition 8.8, and the remarks preceding it, to relate this to $\mathbf{E}(e)$. □

With this relationship between the syntactic and semantic behaviour of head normal forms we can generalise Lemma 8.9:

Proposition 8.11 *For all closed expressions $e \Downarrow s$ if and only if $\mathbf{E}[\![e]\!] \Downarrow s$.*

Proof: The proof follows by induction on the structure of s. The base case is an immediate consequence of Lemma 8.9 so assume $s = a.s'$ for some $a \in Act_{Val}$.

Suppose $e \Downarrow s$. Since the proof system is sound for $\mathbf{E}$, we can assume that e is in head normal form. To prove $\mathbf{E}(e) \Downarrow s$ it is sufficient to prove that $T \Downarrow s'$ where T is such that $\mathbf{E}[\![hnf(e)]\!] \xrightarrow{a} T$. By the previous Lemma we know that T has the structure $\mathbf{E}[\![\sum \{ e' \mid e \xrightarrow{\tau}{}^{*} \xrightarrow{a} e' \}]\!]$. From $e \Downarrow s$ it follows that $e' \Downarrow s'$ for all e' such that $e \xrightarrow{\tau}{}^{*} \xrightarrow{a} e'$; this in turn implies that $\sum \{ e' \mid e \xrightarrow{\tau}{}^{*} \xrightarrow{a} e' \} \Downarrow s'$ and so by induction $T \Downarrow s'$.

Conversely suppose $\mathbf{E}[\![e]\!] \Downarrow s$. To show that $e \Downarrow s$ it is sufficient to prove $e' \Downarrow s'$ for any e' such that $e \xrightarrow{\tau}{}^{*} \xrightarrow{a} e'$. By the previous Lemma $\mathbf{E}[\![hnf(e)]\!] \xrightarrow{a} T$ where $T = \mathbf{E}[\![\sum \{ e' \mid e \xrightarrow{\tau}{}^{*} \xrightarrow{a} e' \}]\!]$. Since $T \Downarrow s$ and $T \leq \mathbf{E}[\![e']\!]$ we have $\mathbf{E}[\![e'']\!] \Downarrow s'$ and now $e'' \Downarrow s'$ follows by induction. □

Proposition 8.12 *For all closed expressions and for every $s \in S$, $e \Downarrow s$ implies $\mathcal{A}(\mathbf{E}[\![e]\!], s) = c(\mathcal{A}(e, s))$.*

Proof: The proof is by induction on the structure of s. Since $e \Downarrow s$ we may assume e is in hnf and we can use again the technique of pushing $\mathbf{E}[\![\ \ ]\!]$ through e so that the top-level is expressed in terms of operators on $\mathbf{D}$.

- Suppose $s = \varepsilon$.

  We need to show that $c(\mathcal{A}(e, \varepsilon)) = \mathcal{A}(\mathbf{E}[\![e]\!], \varepsilon)$. Note that in [11] this result is an immediate corollary of the definition of head-normal forms. Here we need to take into account the effect of the retract $\eta^{*^D}_{\mathbf{D}}$.

  The interesting case here is when $\surd \in |\ \mathcal{A}(e)\ |$. Suppose $e = \sum \{ e_A \mid A \in \mathcal{A} \}$, $e_{\surd} = \sum e_v \mid v$ where each e_v is of the form $\sum \{ e_B \mid B \in \mathcal{B}_v \}$ and suppose $\mathcal{A} = \{A_1, \cdots, A_n\}$. Let $\mathcal{Q}$ be defined by

  $$\mathcal{Q} = \bigcup \{ \mathcal{B}_v \vee \{\surd\} \}.$$

  Rearranging $\mathcal{A}$ so that $\surd \notin A_i$ for $1 \leq i \leq j$ and $\surd \in A_i$ for $j < i \leq n$ and setting

  $$\mathcal{R} = A_1 \cup \cdots \cup A_j \cup ((A_{j+1} \setminus \{\surd\} \vee \mathcal{Q}) \cup \mathcal{Q}) \cup \cdots \cup ((A_n \setminus \{\surd\} \vee \mathcal{Q}) \cup \mathcal{Q})$$

  it is easy to show that $\mathcal{A}(\mathbf{E}[\![e]\!], s) = c(\mathcal{R})$, from the definition of $\oplus_{\mathbf{D}}, +_{\mathbf{D}}, |_{\mathbf{D}}$ and the interaction of c with $\cup$ and $\vee$. So to establish the base case it is now sufficient to show that $\mathcal{R} \subseteq c(\mathcal{A}(e, s))$ and $\mathcal{A}(e, s) \subseteq c(\mathcal{R})$.

  To establish $\mathcal{R} \subseteq c(\mathcal{A}(e, s))$ we need to show that whenever $X \in \mathcal{R}$ then there exists e' such that $e \stackrel{\varepsilon}{\Longrightarrow} e'$ and $\mathcal{S}(e') = X$; as an example consider $X = A_k \setminus \{\surd\} \cup B \cup \{\surd\}$ where $B \cup \{\surd\} \in \mathcal{Q}$ and $j < k \leq n$. From the structure of e we know that $e \stackrel{\varepsilon}{\Longrightarrow} e_{A_k} + e_B \mid v$ and $\mathcal{S}(e_{A_k} + e_B \mid v) = A_k \setminus \{\surd\} \cup B \cup \{\surd\}$.

  To show the converse, that $\mathcal{A}(e, s) \subseteq c(\mathcal{R})$, first it is easy to check that $\mathcal{S}(e) \subseteq c(\mathcal{R})$ and therefore it is sufficient to prove that $\mathcal{S}(e') \in \mathcal{R}$ for any e' such that $e \stackrel{\varepsilon}{\Longrightarrow} e' \stackrel{\tau}{\not\longrightarrow}$. This follows easily from the structure of the two possibilities for e', $\sum \{ e_a \mid a \in A \}$ with $\surd \notin A$ or $\sum \{ e_b \mid b \in B \} \mid v$. This completes the base case.

- Suppose $s = a.s'$. We argue as follows

  $$\begin{aligned}
  c(\mathcal{A}(e, s)) &= c(\bigcup \{ \mathcal{A}(e', s') \mid e \xrightarrow{\tau}{}^{*} \xrightarrow{a} e' \}) \\
  &= c(\mathcal{A}(\sum \{ e' \mid e \xrightarrow{\tau}{}^{*} \xrightarrow{a} e' \}, s'))) \\
  &= \mathcal{A}(T, s') \text{ by induction and Lemma 8.10} \\
  &\quad \text{where } \mathbf{E}[\![e]\!] \xrightarrow{a} T \\
  &= \mathcal{A}(\mathbf{E}[\![e]\!], a.s') \\
  &= \mathcal{A}(\mathbf{E}[\![e]\!], s).
  \end{aligned}$$

□

By composing all of the results in this section we obtain a proof of Theorem 7.6:

Theorem 8.13 *The model* $\mathbf{E}$ *is fully-abstract with respect to the testing preorder* $\sqsubseteq$*, i.e. for all closed expressions* $e \sqsubseteq e'$ *if and only if* $\mathbf{E}[\![e]\!] \leq \mathbf{E}[\![e']\!]$. □

9 Related Work

There has already been a number of attempts at giving an operational semantics for *CML* , or rather core subsets of *CML* but as far as we are aware very few of these have been used to develop a semantic theory or denotational model for the language. For example in [18, 2] the core language λ_{cv} is given a two-level operational semantics which results in a reduction relation between multisets of language expressions. Although this gives a formal semantics which may be referenced by implementors it is insufficient as the basis of a behavioural theory. A similar approach is taken in [7] where a hierarchy of languages is defined and each is given a bisimulation semantics. Starting with a *CCS* like language in which the parallel operator has been replaced with a *fork* operator for process creation, restriction, guarded choice and finally private channel names are added. This last refinement produces a language which is more reminiscent of the $\pi - calculus$ and in particular it does not include any notion of the production of values.

More recently in [3] an operational semantics is given to a language called *FPI* which has many of the programming constructs of *CML*. However it lacks any *spawn* or *fork* construct and indeed later in the same thesis the author notes that in order to accommodate such an operator the operational semantics would have to be modified considerably. Furthermore the operational semantics of value production within the context of the parallel operator in not consistent with that of CML in [18]. In the same thesis a denotational semantics, based on Acceptance Trees, is given for a language very similar to λ_{cv}. However this is not in the style of a semantic function

$$\mathcal{D}: \lambda_{cv} \longrightarrow \mathbf{D}$$

where $\mathbf{D}$ is a semantic domain; instead there is an extra parameter which is defined in terms of a notion of "dynamic types". There is also no result corresponding to the previous Theorem, the *full-abstraction* result, as there is no behavioural semantics given for λ_{cv}.

There has also been some related work on *higher-order processes languages* such as *CHOCS*, [20] and *FACILE*, [5]. Again here the main theoretical emphasis has been on the development of operational semantics in terms of labelled transition systems and to a certain extent the investigation of appropriate versions of bisimulation equivalence for these languages. Another strand of research has been on the development of *type systems* for these kinds of languages, [16], and we will certainly need to build on such work if we are to extend our results to languages which include more of the features of *CML*.

A Operational Semantics for Contexts

In this appendix we develop the operational semantics for contexts which is necessary for the proof of Proposition 5.8.

We can define contexts using the following grammar:

$$\begin{aligned} C \quad ::= \quad & e,\ e \in CPExp \mid [\,] \mid let\ x = C\ in\ C \mid b \mapsto C, C' \mid \\ & spawn(C) \mid n?\lambda x.C \mid n!d.C \mid C + C \mid \\ & C \oplus C \mid\ C \mid C\ \mid local\ n\ in\ C\ end \end{aligned}$$

We say C is a context, often written as $C[\,]$, if it can be generated from the this grammar and in addition it contains at most one occurrence of $[\,]$. Throughout the remainder of this appendix we assume that all $e, e', \ldots$ are closed expressions from *PExp* and all contexts are closed, i.e contain no free occurrences of variables.

We can give an operational semantics to contexts using the notion of *action transducer* as defined in [12]. Transitions are of the form $C \xrightarrow[\lambda]{\mu} C'$, where $\mu \in Act_{Val_\tau}$ and $\lambda \in Act_\tau$, and intuitively this can be interpreted as: whenever $e \xrightarrow{\lambda} e'$ then $C[e] \xrightarrow{\mu} C'[e']$. However sometimes in a move of an expression of the form $C[e]$ there are no contributions from e and therefore in addition we have moves of the form $C \xrightarrow{\mu} C'$ to indicate that $C[e] \xrightarrow{\mu} C[e]$ for any expression e.

The rules defining these transitions are given in Figure 1, where λ ranges over Act_τ, μ over Act_{Val_τ} and γ over $Act_{Val_\tau} \cup \{\cdot\}$; for convenience some obvious symmetric rules for $+$ have been omitted.

We first state some elementary properties of these transitions. Let $C[\,] \downarrow_{[\,]}$ denote that $[\,]$ occurs beneath some prefix in C and $C[\,] \uparrow_{[\,]}$ the converse.

Lemma A.1

1. *If* $C[\,] \xrightarrow[\tau]{\lambda} C'[\,]$ *then* $\lambda = \tau, C'[\,]$ *is* $C[\,]$ *and* $C[\,] \uparrow_{[\,]}$

2. *If* $C[\,] \xrightarrow[\lambda']{\lambda} C'[\,]$ *then* $C[\,] \uparrow_{[\,]}$ *and* $C'[\,] \uparrow_{[\,]}$.

3. $C \uparrow_{[\,]}$ *and* $e \xrightarrow{\lambda} e'$ *implies* $C[e] \xrightarrow{\lambda} C[e']$

Proof: The first two statements are proved by rule induction while the last is by induction on the structure of $C[\,]$. □

We now show that these transitions are consistent with the operational semantics for expressions given in Section 3. This consists in the ability to decompose a move from an expression of the form $C[e]$ into a transition from the context $C[\,]$ and an associated move from e, and a corresponding composition result, composing a transition from $C[\,]$ and an appropriate move from the expression e into a move from the expression $C[e]$.

Lemma A.2 (Move Composition) *For every* $e \in CPExp$

1. $C[\,] \xrightarrow{\lambda} C'[\,]$ *implies* $C[e] \xrightarrow{\lambda} C'[e]$.

2. $C[\,] \xrightarrow[\lambda']{\lambda} C'[\,]$ *and* $e \xrightarrow{\lambda'} e'$ *implies* $C[e] \xrightarrow{\lambda} C'[e']$.

Proof: By induction on the proof of the transitions $C[\,] \xrightarrow{\lambda} C'[\,]$ and $C[\,] \xrightarrow[\lambda']{\lambda} C'[\,]$ respectively. □

Lemma A.3 (Move Decomposition) *If* $C[e] \xrightarrow{\lambda} e''$ *then* e'' *has the form* $C'[e']$ *for some context* $C[\,]$ *and expression* e' *such that either*

1. $C[\,] \xrightarrow{\lambda} C'[\,]$ *and* e *is* e', *or*

2. $C[\,] \xrightarrow[\lambda']{\lambda} C'[\,]$ *and* $e \xrightarrow{\lambda'} e'$.

Proof: By structural induction on $C[\,]$. □

These two results about single transitions are now generalised to composition and decomposition results about sequences of transitions. Their exact formulation are complicated by the fact that in the proof of Proposition 5.8 we will only be able to match up *weak* sequences of derivations.

$$[\,] \xrightarrow[\mu]{\mu} [\,] \qquad \frac{e \xrightarrow{\lambda} e'}{e \xrightarrow{\lambda} e'}$$

$$\frac{C[\,] \xrightarrow[\gamma]{\surd v} C'[\,]}{e \mid C[\,] \xrightarrow[\gamma]{\surd v} e \mid C'[\,]}$$

$$\frac{C[\,] \xrightarrow[\lambda]{\surd v} C'[\,]}{let\ x = C[\,]\ in\ e \xrightarrow[\lambda]{\tau} C'[\,] \mid e} \qquad \frac{C[\,] \xrightarrow[\gamma]{\mu} C'[\,]}{let\ x = C[\,]\ in\ e \xrightarrow[\gamma]{\mu} let\ x = C'[\,]\ in\ e}$$

$$\frac{C[\,] \xrightarrow[\lambda]{\lambda} C'[\,]}{spawn(C[\,]) \xrightarrow[\lambda]{\lambda} spawn(C'[\,])} \qquad \frac{}{spawn(C[\,]) \xrightarrow{\tau} C[\,] \mid null}$$

$$\frac{e \xrightarrow{\surd v} e'}{e + C[\,] \xrightarrow{\surd v} e'} \qquad \frac{C[\,] \xrightarrow[\lambda]{\surd v} C'[\,]}{e + C[\,] \xrightarrow[C]{\surd v} [\,]'}$$

$$\frac{C[\,] \xrightarrow[\lambda]{\tau} C'[\,]}{C[\,] + e \xrightarrow[\lambda]{\tau} C'[\,] + e} \qquad \frac{e \xrightarrow{\tau} e'}{C[\,] + e \xrightarrow{\tau} C[\,] + e'}$$

$$\frac{}{C[\,] \oplus e \xrightarrow{\tau} C[\,]} \qquad \frac{}{C[\,] \oplus e \xrightarrow{\tau} e}$$

$$\frac{}{n?\lambda x.C[\,] \xrightarrow{n?v} C[\,][v/x]} \qquad \frac{}{n!v.C[\,] \xrightarrow{n!v} C[\,]}$$

$$\frac{C[\,] \xrightarrow[\gamma]{\lambda} C'[\,],\ \ n \text{ not in } \lambda}{local\ n\ in\ C[\,]\ end \xrightarrow[\gamma]{\lambda} local\ n\ in\ C'[\,]\ end}$$

$$\frac{C[\,] \xrightarrow[\lambda]{n?v} C'[\,],\ \ e \xrightarrow{n!v} e'}{C[\,] \mid e \xrightarrow[\lambda]{\tau} C'[\,] \mid e'} \qquad \frac{C[\,] \xrightarrow[\lambda]{n!v} C'[\,],\ \ e \xrightarrow{n?v} e'}{C[\,] \mid e \xrightarrow[\lambda]{\tau} C'[\,] \mid e'}$$

Table 1: Operational Rules for Contexts

Proposition A.4 (Derivation Composition) *Suppose $C_0[\,], C_1[\,], \cdots, C_k[\,]$ is a sequence of contexts such that either $C_i[\,] \xrightarrow[\gamma_i]{\lambda_i} C_{i+1}[\,]$ or $C_i[\,]$ is $C_{i+1}[\,]$. Furthermore suppose $e_0, e_1, \cdots, e_k$ is a sequence of closed terms such that*

1. $C_{i+1}[\,]$ *is* $C_i[\,]$ *implies* $C_i[\,]\uparrow_{[\,]}$ *and* $e_i \xrightarrow{\tau} e_{i+1}$.

2. $C_i[\,] \xrightarrow{\lambda_i} C_{i+1}[\,]$ *implies* e_{i+1} *is* e_i.

3. $C_i[\,] \xrightarrow[\lambda_i' \neq \tau]{\lambda_i} C_{i+1}[\,]$ *implies* $e_i \xrightarrow{\lambda_i'} e_{i+1}$.

4. $C_i[\,] \xrightarrow[\tau]{\lambda_i} C_{i+1}[\,]$ *implies* $e_i \xrightarrow{\tau} e_{i+1}$ *or* $e_{i+1} = e_i$.

Then

$$C_0[e_0] \xrightarrow{\lambda_0} C_1[e_1] \cdots \xrightarrow{\lambda_{k-1}} C_k[e_k]$$

is a derivation, where some $\xrightarrow{\lambda_i}$ *may actually be the identity relation.*

Proof: The proof is by induction on k. If $k = 0$ then $C_0[e_0]$ is trivially a derivation. So suppose $k = i + 1$, by induction

$$C_0[e_0] \xrightarrow{\lambda_0} C_1[e_1] \cdots \xrightarrow{\lambda_{k-1}} C_i[e_i]$$

is a derivation and we examine the transition $C_i[\,] \xrightarrow[\gamma_i]{\lambda_i} C_{i+1}[\,]$. There are four cases:

1. Suppose $C_{i+1}[\,]$ is $C_i[\,]$. In this case by premise 1 $C_i\uparrow_{[\,]}$ and $e_i \xrightarrow{\tau} e_{i+1}$. By Proposition A.1.3 $C_i[e_i] \xrightarrow{\tau} C_i[e_{i+1}]$ which completes the derivation.

2. Suppose $C_i[\,] \xrightarrow{\lambda_i}$. By premise 2, e_{i+1} is e_i. By Lemma A.2.1 $C_i[e_i] \xrightarrow{\lambda_i} C_{i+1}[e_i]$ which completes the derivation.

3. Suppose $C_i[\,] \xrightarrow[\lambda_i' \neq \tau]{\lambda_i} C_{i+1}[\,]$. By premise 3, $e_i \xrightarrow{\lambda_i} e_{i+1}$ and by Lemma A.2.2 $C_i[e_i] \xrightarrow{\lambda_i} C_{i+1}[e_{i+1}]$ and the derivation is completed.

4. Finally suppose $C_i[\,] \xrightarrow[\tau]{\lambda_i} C_{i+1}[\,]$ in which case by premise 4 either $e_i \xrightarrow{\tau} e_{i+1}$ and by Lemma A.2.2 $C_i[e_i] \xrightarrow{\lambda_i} C_{i+1}[e_{i+1}]$, or else e_i is e_{i+1}. By Proposition A.1.1 $C_i[\,]$ is $C_{i+1}[\,]$ in which case the transition $\xrightarrow{\lambda_i}$ is the identity and in both cases we have extended the derivation.

□

Proposition A.5 (Derivation Decomposition) *Suppose* $C_0[e_0] \xrightarrow{\lambda_0} e_1' \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_{k-1}} C_k[e_k]$ *is a derivation. Then for all* $0 \leq i \leq k-1$ *there exists a context* $C_i[\,]$, *an expression* e_i *and a* γ_i *such that* e_i' *has the form* $C[e_i]$, $C_i[\,] \xrightarrow[\gamma_i]{\lambda_i} C_{i+1}[\,]$ *and*

1. $\gamma_i = \cdot$ *and* e_{i+1} *is* e_i, *or*

2. *or* $\gamma_i \neq \cdot$ *and* $e_i \xrightarrow{\gamma_i} e_{i+1}$.

Proof: By induction on k using Lemma A.2. □

We are now ready to present the main result of the Appendix:

Theorem A.6 *Suppose $e \ll e'$ and $C[e]$ must w, where w does not occur in e, e'. Then $C[e']$ must w.*

Proof: Consider an arbitrary computation from $C[e']$

$$C[e'] \equiv C_0[e'_0] \xrightarrow{\tau} e''_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} e''_k \xrightarrow{\tau} \ldots\ldots \tag{5}$$

First let us assume that this is a finite computation, i.e. there exists some n such that $e''_n \stackrel{\tau}{\not\longrightarrow}$. We need to show that there exists $0 \leq i \leq n$ such that $e''_i \xrightarrow{\surd w}$. The basic idea of the proof is to decompose this computation into a contribution from e' and a contribution from the context $C[\,]$, use the fact that $e \ll e'$ to find a similar contribution from e and recombine it with the contribution from $C[\,]$ to obtain a computation from $C[e]$ which uses exactly the same contexts. This will ensure that $e''_i[\,] \xrightarrow{\surd w}$ for some i.

Applying the Derivation Decomposition result, Proposition A.5, to (5) we get sequences $C[\,]$, e'_i, for $0 \leq i \leq n-1$, such that e''_i has the form $C_i[e'_i]$, $C_i[\,] \xrightarrow[\gamma_i]{\tau} C_{i+1}[\,]$, with corresponding properties for e'_i. We will show that there is a maximal computation from $C[e]$ which uses only the contexts $C[\,]$, thereby ensuring that $C_i[\,] \xrightarrow{\surd w}$ for some i. There are two cases, according to whether or not the entire computation is independent of e'.

1. $\gamma_i = \cdot$ for all $0 \leq i \leq n-1$. This includes the case when $n = 0$, i.e. $C_0[e'_0] \stackrel{\tau}{\not\longrightarrow}$. By repeated application of the Move Composition result Lemma A.2.1, we get

   $$C[e] \equiv C_0[e] \xrightarrow{\tau} \cdots \xrightarrow{\tau} C_n[e]$$

   which also implies that e'''_n is e'. However this derivation may not be maximal but we show that it can always be extended to a maximal computation using exactly the same contexts.

   Here we know that each e'_i coincides with e' and therefore since $C_n[e'] \stackrel{\tau}{\not\longrightarrow}$ we can assume that $C_n[\,] \uparrow_{[\,]}$, $e' \stackrel{\tau}{\not\longrightarrow}$ and $C_n[\,] \xrightarrow[\alpha]{\tau}$ for no $\alpha \in \mathcal{S}(e')$. There are two sub-cases:

   (a) $e \Uparrow$ so that $e \xrightarrow{\tau} e^1 \xrightarrow{\tau} e^2 \cdots$. By Proposition A.1.3 we have

   $$C_n[e] \xrightarrow{\tau} C_n[e^1] \xrightarrow{\tau} \cdots$$

   which is an infinite computation of the required form; i.e. it only uses the contexts $C_i[\,]$ and since $C[e]$ *must* w there exists $0 \leq i \leq n-1$ such that $C_i[e] \xrightarrow{\surd w}$.

   (b) $e \Downarrow$. Since $e' \stackrel{\tau}{\not\longrightarrow}$ and $e \ll e'$ we know that there exists g such that $e \stackrel{\varepsilon}{\Longrightarrow} g \stackrel{\tau}{\not\longrightarrow}$ and $\mathcal{S}(g) \subseteq \mathcal{S}(e)$. Again by Proposition A.1.3 we have

   $$C_n[e] \xrightarrow{\tau} \cdots \xrightarrow{\tau} C_n[e'] \stackrel{\tau}{\not\longrightarrow}$$

   We therefore, again, have a maximal computation of the required form and the result follows because $C[e]$ *must* w.

2. There exists i such that $\gamma_i \neq \cdot$. Let t denote the sequence of elements in $\gamma_0 \cdots \gamma_{n-1}$ which are different from $\cdot$ and let s the result of removing all τ actions from t. There are two possibilities

   (a) $e \Uparrow s$ in which case

   $$e = e_0 \xrightarrow{\mu_0} e_1 \xrightarrow{\mu_1} \cdots e_{m-1} \xrightarrow{\mu_{m-1}} e_m \Uparrow$$

   where $\mu_0 \cdots \mu_{m-1}$ is a prefix of t. There are two further sub-cases:

   i. $m = 0$. Let i be the smallest index such that $\gamma_i \neq \cdot$, which is guaranteed to exist. By repeated use of the Move Composition result, Lemma A.2.1, we get the sequence

   $$C_0[e] \xrightarrow{\tau} \cdots \xrightarrow{\tau} C_i[e].$$

   Since $C_i[\,] \xrightarrow[\gamma_i]{\tau}$ where γ_i is different from $\cdot$ Proposition A.1.2 ensures that $C_i[\,] \uparrow_{[\,]}$ and by Lemma A.1.3 we get the infinite computation

$C_i[e] \xrightarrow{\tau} C_i[e^1] \xrightarrow{\tau} \cdots$

which again is of the required form.

ii. $m > 0$. Here without loss of generality we may assume $\mu_{m-1} \neq \tau$. We can almost build a computation from $C[e]$, except that some τ steps in $\mu_0 \cdots \mu_{m-1}$ and $\gamma_0 \cdots \gamma_{n-1}$ may not match. We can identify two possibilities:

A. for some $j, C_j[\,] \xrightarrow[\gamma_j]{\tau} C_{j+1}[\,]$ where $\gamma_i \neq \cdot$ and $e_j \xrightarrow{\tau^*} e'_j \xrightarrow{\gamma_j} e_{j+1}$. By Lemma A.1.2 $C_j[\,] \uparrow_{[\,]}$ and therefore we can construct the sequence of moves

$C_j[e_j] \xrightarrow{\tau} \cdots \xrightarrow{\tau} C_j[e'_j] \xrightarrow{\tau} C_{j+1}[e_{j+1}]$

using Lemma A.1.3.

B. for some $j, C_j[\,] \xrightarrow[\tau]{\tau} C_{j+1}[\,]$ but $e_j \not\xrightarrow{\tau}$. In this case we can let $e_{j+1} = e_j$, since by Proposition A.1.1 $C_j[\,]$ is $C_{j+1}[\,]$.

Repeating these steps where necessary we can build a sequence of contexts $D_0[\,] \cdots D_m[\,]$ where for each i, $0 \leq i \leq m$ there exists j, $0 \leq j \leq n-1$ such that $D_i[\,] \equiv C_j[\,]$ and a sequence of closed terms $e_0 \cdots e_m$ both of which satisfy the premises of Derivation Composition result Proposition A.4. In which case we have a derivation

$C[e] \equiv D_0[e_0] \xrightarrow{\tau} \cdots \xrightarrow{\tau} D_m[e_m]$.

Since $\mu_{m-1} \neq \tau$ then by Lemma A.1.2 we have that $D_m[\,] \uparrow_{[\,]}$ which implies that, because $e_m \Uparrow$, we can construct the infinite sequence

$D_m[e_m] \xrightarrow{\tau} D_m[e^1_m] \xrightarrow{\tau} \cdots$

This again means we have constructed a maximal computation of the required form from $C[e]$.

(b) $e \Downarrow s$. Here we know $e' \Downarrow s$ and there exists some e_n such that $e \stackrel{s}{\Longrightarrow} e_n$, $e_n \not\xrightarrow{\tau}$ and $\mathcal{S}(e_n) \subseteq \mathcal{S}(e'_n)$.

Therefore we can proceed as in the previous sub-case to construct a maximal computation of the required form:

$C[e] \equiv C_0[e] \xrightarrow{\tau} \cdots \xrightarrow{\tau} C_n[e_n]$.

It remains to consider the case when the initial computation, (5), is infinite. However our context have been defined to be finite, i.e. have no occurrence of the recursion operator, and therefore one can show that there must exist some n such that for all $k > n$ the k^{th} element of the computation has the form $C_n[e_k]$ where $e_{k-1} \xrightarrow{\tau} e_k$. In particular this means that $e_n \Uparrow$. The argument now proceeds as in case 2 (a) above. □

B Definition of Operators on D

This appendix describes the remaining continuous functions op_D over domain D which were omitted in Section 7.

Local

Define $local\ n\ in\ end_{\mathbf{D}} : \mathbf{D} \longrightarrow \mathbf{D}$ by $Y\lambda X.\Phi$ where

$\Phi = fold \circ up(R\ X) \circ unfold$ and

$R\ X\ \langle \mathcal{A}, f \rangle = \langle \mathcal{A} \setminus \{n!, n?\}, X \circ f\ |_{\mathcal{A}\setminus\{n!,n?\}} \rangle$

Output

Define $out_{\mathbf{D}} : Chan \times Val \times \mathbf{D} \longrightarrow \mathbf{D}$ by

$out_{\mathbf{D}}\ n\ v\ d = fold(\{\{n!\}\}, \{v \mapsto d\})$

Tick

This operator is introduced only to give a more succinct definition of $|_{\mathbf{D}}$.

Define $\surd : Val \longrightarrow \mathbf{D} \longrightarrow \mathbf{D}$ by

$\surd\ v\ d = fold(\langle\{\{\surd\}\}, \{v \mapsto d\}\rangle)$

Parallel

Define $|_{\mathbf{D}}: \mathbf{D} \times \mathbf{D} \longrightarrow \mathbf{D}$ by $Y\lambda X.\Phi$ where

$\Phi = fold(up(R\ X))$ and

$$R(\langle \mathcal{A}, f\rangle, \langle \mathcal{B}, g\rangle) = \sum\!\!\!\!\!\bigcirc \{\, T_{AB} \mid A \in \mathcal{A}, B \in \mathcal{B} \,\}$$

where

$$\begin{array}{rcl} t_{AB} & = & \text{if } INT(A,B) = \emptyset \\ & & \text{then } sumext(A,B) \\ & & \text{else } (sumext(A,B) + sumint(A,B)) \oplus sumint(A,B) \end{array}$$

and

$$\begin{array}{rcl} sumext(A,B) & = & \sum\{EXT(A,B)\} \\ sumint(A,B) & = & \sum\!\!\!\!\!\bigcirc\, \{INT(A,B)\} \\ INT(A,B) & = & \{\, X(f(n?)(v), g(n!)(v) \mid n? \in A, n! \in B, v \in dom(g(n!))\,\} \cup \\ & & \{\, X(f(n!)(v), g(n?)(v) \mid n! \in A, n? \in B, v \in dom(f(n!))\,\} \\ EXT(A,B) & = & \{\, in_D(n, \lambda v.X(f(n?)(v), d_2) \mid n? \in A\,\} \cup \\ & & \{\, in_D(n, \lambda v.X(d_1, g(c?)(v)) \mid n? \in A\,\} \cup \\ & & \{\, \surd(v, X(d_1, g(\surd)(v)) \mid \surd \in B, v \in dom(g(\surd))\,\} \cup \\ & & \{\, out_D(n, v, X(f(n!)(v), d_2)) \mid n! \in A, v \in dom(f(n!))\,\} \cup \\ & & \{\, out_D(n, v, X(d_1, g(n!)(v))) \mid n! \in B, v \in dom(g(n!))\,\} \end{array}$$

References

[1] J.C.M. Baeten and F.W. Vaandrager. An algebra for process creation. *Acta Informatica*, 29(4):303–334, 1992.

[2] R. Milner D. Berry and D. Turner. A Semantics for ML Concurrency Primitives. In *Proceedings of the 19th ACM Symposium on Principles of Programmings Languages*, 1992.

[3] Mourad Debbabi. *Integration des Paradigmes de Programmation Parallele, Fonctionnelle et Imperative: Fondement Semantiques.* Phd thesis, Universite D'Orsay, 1994.

[4] William Ferreira. *Towards a Semantic Theory of CML.* Phd thesis, Univsity of Sussex, 1995.

[5] A. Giacalone, P. Mistra, and S. Prasad. Operational and algebraic semantics for facile: A symmetric integration of concurrent and functional programming. In *Proceedings of ICALP 90*, volume 443 of *Lecture Notes in Computer Science*, pages 765–780. Springer-Verlag, 1990.

[6] Carl A. Gunter. *Semantics of Programming Languages.* MIT Press, Cambridge Massachusetts, 1992.

[7] K. Havelund. *The Fork Calculus: Towards a Logic for Concurrent ML.* PhD thesis, Ecole Normale Superieur, Paris, 1994.

[8] M. Hennessy. *Algebraic Theory of Processes.* MIT Press, Cambridge, Massachusetts, 1988.

[9] M. Hennessy and A. Ingolfsdottir. A theory of communicating processes with value-passing. *Information and Computation*, 107(2):202–236, 1993.

[10] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, Englewood Cliffs, 1985.

[11] A. Ingolfdottir. *Semantic Models for Communicating Processes with Value Passing.* PhD thesis, University of Sussex, 1994.

[12] K. Larsen. Compositional Theories based on an Operational Semantics of Contexts. Technical Report R 89-32, University of Aalborg, Department of Mathematics and Computer Science, Sepember 1989.

[13] R. Milner. *Communication and Concurrency.* Prentice-Hall International, Englewood Cliffs, 1989.

[14] E. Moggi. Computational Lambda Calculus and Monads. Report ECS-LFCS-88-66, Edinburgh LFCS, 1988.

[15] J.H. Morris. *Lambda Calculus Models of Programming Languages.* PhD thesis, M.I.T., 1968.

[16] F. Nielson and H.R. Nielson. From cml to process algebras. Report DAIMI PB-433, University of Arhus, 1993.

[17] G.D. Plotkin. Lecture notes in domain theory, 1981. University of Edinburgh.

[18] John Reppy. *Higher-Order Concurrency.* PhD thesis, Cornell University, June 1992. Technical Report TR 92-1285.

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

[20] B. Thomsen. *Calculi for Higher-Order Communicating Systems.* Phd thesis, Imperial College, 1990.