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

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 he 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

- $n?\lambda x.e$, 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

(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 $let\ x = e_1$

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

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

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

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

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

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

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

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

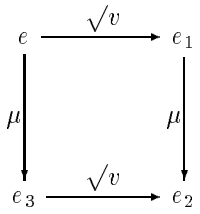
$(BoolA)$ $\dfrac{e_1 \xrightarrow{\mu} e_1', [\![b]\!] = true}{b \mapsto e_1, e_2 \xrightarrow{\mu} e_1'}$ $\qquad$ $\dfrac{e_2 \xrightarrow{\mu} e_2', [\![b]\!] = false}{b \mapsto e_1, e_2 \xrightarrow{\mu} e_2'}$

(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) $\qquad e$

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

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

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 $S = Act^* \cup \{\, s_1 \surd v s_2 \mid v \in Val, s_1, s\ ^*$

$\rangle\lceil\in\in\triangle\{\Rightarrow\langle\Rightarrow$

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

Proposition 5.8

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 rees

Here we first review the version of Acceptance Trees [illegible]

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 $PExp$

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

$$in_{\mathbf{D}}\ n\ f = fold\,\langle\{\{n?\}\}, n? \mapsto f\rangle\rangle$$

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

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

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

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

$$\frac{}{let\ rec\ P}$$

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 α_i, β

Suppose $X = \sum_{i \in I} \alpha_i . X_i$

□

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}.

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 compositiontheexpressionmpnoa

$$[\,] \xrightarrow[\mu]{\mu} [\,]$$

$$\frac{e \xrightarrow{\lambda} e'}{e \xrightarrow{\lambda} e'}$$

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

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

$$\frac{C[\,] \xrightarrow[\gamma]{\mu} C'[\,]}{let\ x = C[\,]\ in\ e \xrightarrow[\gamma]{\mu} let\ x = C'[\,]\ in\ e}$$

$$\frac{C[\,] \xrightarrow[\lambda]{\lambda} C'[\,]}{spawn(C[\,]) \xrightarrow[\lambda]{\lambda} spawn(C'[\,])}$$

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

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

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

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

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

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

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

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

$$\frac{}{n!v.C[\,] \xrightarrow{n!v} C[\,]}$$

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

$$C[\,]$$

1. C_{i+1}

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

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

$C_j[e_j$

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 $|_{\mathbf{D}}\colon \mathbf{D} \times \mathbf{D} \longrightarrow \mathbf{D}$ 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

$$\begin{array}{rcl} sumext(A,B) & = & \sum\{EXT(A,B)\} \\ sumint(A,B) & = & \sum\!\!\!\!\!\!\bigcirc\, \{INT(A,B)\} \\ INT(A,B) & = & \{\, X(f(n?)(v), g(n!)(v) \mid n? \in A, n! \in B, v \in dom(g(n!))\,\} \cup \\ & & \{\, X(f(n!)(v), g(n?)(v) \mid n! \in A, n? \in B, v \in dom(f(n!))\,\} \\ EXT(A,B) & = & \{\, in_D(n, \lambda v.X(f(n?)(v), d_2) \mid n? \in A\,\} \cup \\ & & \{\, in_D(n, \lambda v.X(d_1, g(c?)(v)) \mid n? \in A\,\} \cup \\ & & \{\, \not{in}_D(n, \lambda v.X(\end{array}$$

[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.