# A fully abstract semantics for a concurrent functional language with monadic types

Alan Jeffrey
School of Cognitive and Computing Sciences
University of Sussex, Brighton BN1 9QH, UK
*alanje@cogs.susx.ac.uk*

**Abstract**

*This paper presents a typed higher-order concurrent functional programming language, based on Moggi's monadic metalanguage and Reppy's Concurrent ML. We present an operational semantics for the language, and show that a higher-order variant of the traces model is fully abstract for may-testing. This proof uses a program logic based on Hennessy–Milner logic and Abramsky's domain theory in logical form.*

## 1 Introduction

This paper presents an operational semantics for a concurrent functional programming language, based on Reppy's [26, 27] Concurrent ML, and Moggi's [22] monadic metalanguage.

CML is a concurrent extension of New Jersey ML, which adds communication primitives based on CCS [19] and CSP [11]. Reppy introduces a new type constructor of *events*, which can spawn concurrent processes, and communicate with them along channels.

Three of the constructors for the event type are:

$$\begin{aligned}\mathtt{always} &: \alpha \to \alpha\,\mathtt{event}\\ \mathtt{wrap} &: (\alpha\,\mathtt{event} \times \alpha \to \beta) \to (\beta\,\mathtt{event})\\ \mathtt{sync} &: \alpha\,\mathtt{event} \to \alpha\end{aligned}$$

These are:

- $\mathtt{always}\,e$ is an event which always returns $e$,
- $\mathtt{wrap}(e, f)$ is an event which evaluates $e$ and applies $f$ to the result, and
- $\mathtt{sync}\,e$ starts the evaluation of a event.

Moggi has proposed a more radical type system for computation, where *all* computation is indicated in the type system by a type constructor, here denoted $\mathsf{C}$. So whereas $\mathsf{nat}$ in Moggi's setting is a type whose expressions are integers, $\mathsf{C}\,\mathsf{nat}$ is a type whose expressions are computations of integers. For example, 37 is of type $\mathsf{nat}$ where $37 + 52$ is of type $\mathsf{C}\,\mathsf{nat}$.

Moggi has shown that a simple programming language called the monadic metalanguage, equipped with certain equations forms a *strong monad* [16, for example], that is a category $\mathbf{C}$ with a functor $T : \mathbf{C} \to \mathbf{C}$ with natural transformations:

$$\begin{aligned}\eta_X &: X \to TX\\ \mu_X &: T^2X \to TX\\ t_{X,Y} &: X \times TY \to T(X \times Y)\end{aligned}$$

subject to certain commuting diagrams. There is a close correspondence between Reppy's operators for CML and the structure of a monad:

| *CML gadget* | *Monadic gadget* |
|---|---|
| event | $T$ on objects |
| wrap | $T$ on arrows |
| always | $\eta$ |
| sync | $\mu$ |
| $\lambda(x,y)\,.\,\mathtt{wrap}(y, \lambda z\,.\,(x,z))$ | $t$ |

In this paper, we shall show how the practice of CML and the theory of MML can be brought together. We present a concurrent programming language CMML whose type system is based on MML, and whose concurrent features are based on a subset of CMML. We present an operational and denotational semantics for CMML, and show that the denotational semantics is fully abstract.

The new results of this paper are:

- Applying Moggi's theory in an operational, rather than denotational, setting. Moggi has concentrated on the equational and denotational semantics for MML, for example expressing nondeterminism as powerdomain rather than operationally. The use of monads in lazy functional programming, pioneered by Wadler [31] and since used in specifying IO [6, 7] for Haskell [14] has concentrated on the algebraic properties of computation types.
- Providing an operational semantics for a CML-like language in the form of a labelled transition system. Reppy's operational semantics for CML uses a reduction system more like Standard ML [21] than CCS.
- Giving a fully abstract semantics for a typed higher-order concurrent language. Hennessy [10] has proved

full abstraction for untyped higher-order processes, and there has been much work on translating higher-order languages like CHOCS [29, 30] or Facile [4, 28] into basic languages like the $\pi$-calculus [20]. Bolignano and Debabi [5] have presented an operational and denotational semantics, which is more complex than that given here, but do not have a full abstraction result.

This paper is an extended abstract of part of [15]. That paper also contains foundational material on the categorical structure formed by various algebras of programming languages, and a translation of a subset of CML into CMML, which is correct up to weak bisimulation. All proofs are only given in sketch form, and are detailed in the full paper.

## 2 Syntax

The language we shall consider in this paper is a typed functional programming language with concurrent and nondeterministic features.

The type system is based on MML, and includes a computation type constructor $\mathsf{C}\,\tau$.

The communication mechanism is based on CML, however, our treatment is missing a number of features, most importantly dynamic channel creation. This is allowed in CML, and is addressed operationally in languages such as Milner, Parrow and Walker's [20] $\pi$-calculus, and Thomsen's [29, 30] CHOCS. Pitts and Stark [24] have investigated denotational models of such languages, but there is not (yet) a fully abstract model for such languages in **Alg** or a similar category of domains. Since we are interested in showing fully abstraction for the traces semantics, we shall not attempt a treatment dynamic name creation for the moment.

A signature with *booleans, channels and deconstructors* is a signature with:

- a set of *sorts* ranged over by $A$, $B$, and $C$,
- a set of *constructors* ranged over by $c$, with sorting $c : A_1, \ldots, A_n \rightarrow B$,
- a set of *deconstructors* ranged over by $d$, with sorting $d : A_1, \ldots, A_n \rightarrow B$,
- a sort $\mathsf{bool}$ with constructors $\mathsf{true}, \mathsf{false} : \rightarrow \mathsf{chan}$, and
- a sort $\mathsf{chan}$ with deconstructor $\mathsf{eq} : \mathsf{chan}, \mathsf{chan} \rightarrow \mathsf{bool}$ such that the term algebra for $\mathsf{chan}$ is countably in£nite.

a sort $\mathsf{chan}$ such For example, we can de£ne a signature $\mathsf{NatList}$ with sorts:

$$\mathsf{bool} \qquad \mathsf{nat} \qquad \mathsf{list}$$

constructors:

$$\begin{array}{rcl} \mathsf{true}, \mathsf{false} & : & \rightarrow \mathsf{bool} \\ \mathsf{zero} & : & \rightarrow \mathsf{nat} \\ \mathsf{succ} & : & \mathsf{nat} \rightarrow \mathsf{nat} \\ \mathsf{nil} & : & \rightarrow \mathsf{list} \\ \mathsf{cons} & : & \mathsf{nat}, \mathsf{list} \rightarrow \mathsf{list} \end{array}$$

and deconstructors:

$$\begin{array}{rcl} \mathsf{eq} & : & \mathsf{nat}, \mathsf{nat} \rightarrow \mathsf{bool} \\ \mathsf{pred} & : & \mathsf{nat} \rightarrow \mathsf{nat} \\ \mathsf{isnil} & : & \mathsf{list} \rightarrow \mathsf{bool} \\ \mathsf{hd} & : & \mathsf{list} \rightarrow \mathsf{nat} \\ \mathsf{tl} & : & \mathsf{list} \rightarrow \mathsf{list} \end{array}$$

and we can use $\mathsf{nat}$ as the sort of channels.

Let **SigBCD** be the category of signatures with booleans, channels and deconstructors, together with morphisms which respect the booleans, channels, and sorting of constructors and deconstructors.

Given a signature with booleans, channels and deconstructors $\Sigma$, we can de£ne the language $\mathsf{CMML}\,\Sigma$ to be the *concurrent monadic metalanguage* over $\Sigma$ given by the grammar:

$$\begin{array}{rcl} e & ::= & * \mid c(e_1, \ldots, e_n) \mid d(e_1, \ldots, e_n) \mid (e, e) \mid v \\ & & \mid [e] \mid \mathsf{let}\, x \Leftarrow e \,\mathsf{in}\, e \mid \lambda x \,.\, e \mid ee \mid \mathsf{if}\, e \,\mathsf{then}\, e \,\mathsf{else}\, e \\ & & \mid \delta \mid e \,\Box\, e \mid \mathsf{fix}(x = e) \mid e!_\tau e \mid e?_\tau \mid e \parallel e \mid e \upharpoonright \vec{e} \\ v & ::= & x \mid v.\mathrm{L} \mid v.\mathrm{R} \end{array}$$

where $x$ ranges over a set of *variables*. We shall call expressions $v$ *lvalues*. These expressions are:

- $*$ is the only closed term of unit type.
- $c(\vec{e})$ is the application of a constructor, to build a value of base type.
- $d(\vec{e})$ is the application of a deconstructor, to build a computation of base type.
- $(e, f)$ is pairing.
- $v.\mathrm{L}$ and $v.\mathrm{R}$ are the left and right projections. We shall see later that restricting projections to lvalues allows us to know the syntactic form of terms from just the type information.
- $[e]$ is a computation which immediately terminates with result $e$. This is similar to 'exit' in LOTOS [1], and 'return' in CML.
- $\mathsf{let}\, x \Leftarrow e \,\mathsf{in}\, f$ is a computation which evaluates $e$ until it returns a value, which is then bound to $x$ in $f$. For example, $\mathsf{let}\, x \Leftarrow [\mathsf{zero}] \,\mathsf{in}\, [\mathsf{succ}\, x]$ is the same as $[\mathsf{succ}\,\mathsf{zero}]$.
- $\lambda x \,.\, e$ is function binding.
- $e f$ is function application.
- $\delta$ is a deadlocked term, which has no reductions.
- $e \,\Box\, f$ gives the external choice between $e$ and $f$. This is similar to CSP's external choice, and CML's 'choose'.
- $\mathsf{fix}(x = e)$ gives the £xed point of $e$ at $x$.
- $e!_\tau f$ outputs the expression $f$ of type $\tau$ along channel $e$ and then returns $*$. This is similar to CML's 'send'.
- $e?_\tau$ inputs any expression $f$ of type $\tau$ along channel $e$ and then returns $f$. This is similar to CML's 'accept'.
- $e \parallel f$ performs $e$ and $f$ in parallel, allowing them to communicate. It will return any value that $f$ returns. This can be de£ned in terms of CML's 'spawn'.

- $e \upharpoonright f_1, \ldots, f_n$ behaves like $e$ but can only communicate on the channels $f_i$. This is similar to SCCS's [18] restriction, but does not have an analogue in CML, where channel scope is treated as in the π-calculus.

We can give $\mathsf{CMML}\,\Sigma$ a static type system, with types:

$$\tau ::= I \mid [A] \mid \tau \otimes \tau \mid \mathsf{C}\,\tau \mid \tau \rightarrow \mathsf{C}\,\tau$$

These types are:

- $I$ is the unit type, whose only closed term is $*$,
- $\sigma \otimes \tau$ is the type of pairs of $\sigma$ and $\tau$,
- $[A]$ is a base type taken from $\Sigma$,
- $\mathsf{C}\,\tau$ is a computation, which returns an expression of type $\tau$, and
- $\sigma \rightarrow \mathsf{C}\,\tau$ is a function, which when applied to a an expression of type $\sigma$ returns an expression of type $\mathsf{C}\,\tau$.

The type judgements for CMML are of the form $\Gamma \vdash e : \tau$ given by rules in Table 1, where $\Gamma$ ranges over *contexts* of the form $x_1 : \tau_1, \ldots, x_n : \tau_n$.

For example, $\mathsf{succ}\,\mathsf{zero}$ is an expression of type $[\mathsf{nat}]$, whereas $\mathsf{pred}(\mathsf{succ}\,\mathsf{zero})$ is an expression of type $\mathsf{C}[\mathsf{nat}]$. This corresponds to the intuition that whereas 1 is *data*, and can be stored in an implementation as a bit string, $1-1$ is *computation*, and has to be stored as a pointer to code (until it is evaluated, at which point the result zero is of type $[\mathsf{nat}]$).

Moggi has shown how the call-by-value λ-calculus can be translated into the monadic metalanguage, and the full version of this paper contains a translation of a subset of CML into CMML, which is correct up to weak bisimulation.

Note that we are only allowing functions to return computations, for example there is no type $I \rightarrow I$, only $I \rightarrow \mathsf{C}\,I$. This corresponds to our intuition that the only terms which involve computation are terms of type $\mathsf{C}\,\tau$, and this would not be true if we allowed functions to return arbitrary type.

This restriction, coupled with the restriction of projections $v.\mathrm{L}$ and $v.\mathrm{R}$ to lvalues allows us to show that:

- any term of type $I$ is either an lvalue or $*$,
- any term of type $[A]$ is either an lvalue or of the form $c(e_1, \ldots, e_n)$,
- any term of type $\sigma \otimes \tau$ is either an lvalue or of the form $(e, f)$, and
- any term of type $\sigma \rightarrow \mathsf{C}\,\tau$ is either an lvalue or of the form $\lambda x \,.\, e$.

In particular, the only closed terms of type $[A]$ are taken from the term algebra for $\Sigma$, so $\mathsf{CMML}\,\Sigma$ is a conservative extension of the term algebra for $\Sigma$.

We have only de£ned projections on lvalues, however we know that any term $\Gamma \vdash e : \sigma \otimes \tau$ is either a pair or an lvalue, and so we can de£ne $\Gamma \vdash \pi e : \sigma, \pi' e : \tau$ as syntactic sugar:

$$\begin{array}{rclrcl} \pi v &=& v.\mathrm{L} & \pi(e,f) &=& e \\ \pi' v &=& v.\mathrm{R} & \pi'(e,f) &=& f \end{array}$$

We can use this to de£ne substitution $e[f/x]$ in the normal fashion, except that we substitute for lvalues as:

$$(v.\mathrm{L})[f/x] = \pi(v[f/x]) \quad (v.\mathrm{R})[f/x] = \pi'(v[f/x])$$

We have only de£ned recursion on computations rather than on functions. However, we can de£ne recursive functions as syntactic sugar:

$$\mathsf{fix}(x = \lambda y \,.\, e) = \lfloor \mathsf{fix}(z = (\lambda x \,.\, [\lambda y \,.\, e])\lfloor z \rfloor) \rfloor$$

where:

$$\lfloor e \rfloor = \lambda y \,.\, \mathsf{let}\, x \Leftarrow e \,\mathsf{in}\, xy$$

These have typing:

$$\frac{\Gamma \vdash e : \mathsf{C}(\sigma \rightarrow \mathsf{C}\,\tau)}{\Gamma \vdash \lfloor e \rfloor : \sigma \rightarrow \mathsf{C}\,\tau} \quad \frac{\Gamma, x : \sigma \rightarrow \mathsf{C}\,\tau, y : \sigma \vdash e : \mathsf{C}\,\tau}{\Gamma \vdash \mathsf{fix}(x = \lambda y \,.\, e) : \sigma \rightarrow \mathsf{C}\,\tau}$$

We can also write $\lambda(\vec{x}) \,.\, e$ as syntactic sugar, for example:

$$\lambda(x,y) \,.\, e = \lambda z \,.\, e[z.\mathrm{L}/x, z.\mathrm{R}/y]$$

For example, a recursive function to add an element to the end of a list is:

$$\begin{array}{rcll} \mathsf{snoc} &:& [\mathsf{list}] \otimes [\mathsf{nat}] \rightarrow \mathsf{C}[\mathsf{list}] & \\ \mathsf{snoc} &=& \mathsf{fix}(u = \lambda(v,w) \,. & \mathsf{let}\, x \Leftarrow \mathsf{isnil}\, v \,\mathsf{in} \\ &&& \mathsf{if}\, x \\ &&& \mathsf{then}\, [\mathsf{cons}(w, \mathsf{nil})] \\ &&& \mathsf{else}\, \mathsf{let}\, y \Leftarrow \mathsf{hd}(v) \,\mathsf{in} \\ &&& \quad \mathsf{let}\, y' \Leftarrow \mathsf{tl}(v) \,\mathsf{in} \\ &&& \quad \mathsf{let}\, z \Leftarrow v(y', w) \,\mathsf{in} \\ &&& \quad [\mathsf{cons}(y, z)]) \end{array}$$

and an unbounded buffer can be de£ned by maintaining a list of elements:

$$\begin{array}{rcll} \mathsf{buff} &:& [\mathsf{chan}] \otimes [\mathsf{chan}] \otimes [\mathsf{list}] \rightarrow \mathsf{C}\,\tau & \\ \mathsf{buff} &=& \mathsf{fix}(u = \lambda(x,y,y) \,. & \mathsf{let}\, w \Leftarrow x?_{\mathsf{nat}} \,\mathsf{in} \\ &&& \mathsf{let}\, z \Leftarrow \mathsf{snoc}(z, w) \,\mathsf{in} \\ &&& u(x,y,z) \\ &&& \square\, \mathsf{let}\, w \Leftarrow \mathsf{hd}\, z \,\mathsf{in} \\ &&& \mathsf{let}\, v \Leftarrow y!_{\mathsf{nat}} w \,\mathsf{in} \\ &&& \mathsf{let}\, z \Leftarrow \mathsf{tl}\, z \,\mathsf{in} \\ &&& u(x,y,z)) \end{array}$$

Note that programs in CMML are often more verbose than in CML, due to the number of $\mathsf{let}$ statements required. This is the cost of making the evaluation order syntactically explicit, rather than implicit as in ML.

## 3 Operational semantics

In this section we de£ne the operational semantics of $\mathsf{CMML}\,\Sigma$. This is given as a *higher-order symbolic value production system*, that is:

$$\frac{}{\Gamma \vdash * : I} \qquad \frac{\Gamma \vdash e : \sigma \quad \Gamma \vdash f : \tau}{\Gamma \vdash (e,f) : \sigma \otimes \tau}$$

$$\frac{\Gamma \vdash e_1 : [A_1] \quad \cdots \quad \Gamma \vdash e_n : [A_n]}{\Gamma \vdash c(e_1,\ldots,e_n) : [A]}[c : A_1,\ldots,A_n \to A] \quad \frac{\Gamma \vdash e_1 : [A_1] \quad \cdots \quad \Gamma \vdash e_n : [A_n]}{\Gamma \vdash d(e_1,\ldots,e_n) : \mathsf{C}[A]}[d : A_1,\ldots,A_n \to A]$$

$$\frac{\Gamma \vdash v : (\sigma \otimes \tau)}{\Gamma \vdash v.\mathrm{L} : \sigma} \quad \frac{\Gamma \vdash v : (\sigma \otimes \tau)}{\Gamma \vdash v.\mathrm{R} : \tau} \quad \frac{}{\Gamma, x : \sigma \vdash x : \sigma} \quad \frac{\Gamma \vdash y : \tau}{\Gamma, x : \sigma \vdash y : \tau}[x \neq y]$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash [e] : \mathsf{C}\tau} \quad \frac{\Gamma \vdash e : \mathsf{C}\sigma \quad \Gamma, x : \sigma \vdash f : \mathsf{C}\tau}{\Gamma \vdash \mathsf{let}\, x \Leftarrow e\, \mathsf{in}\, f : \mathsf{C}\tau} \quad \frac{\Gamma, x : \sigma \vdash e : \mathsf{C}\tau}{\Gamma \vdash \lambda x\,.\,e : \sigma \to \mathsf{C}\tau} \quad \frac{\Gamma \vdash e : \sigma \to \mathsf{C}\tau, f : \sigma}{\Gamma \vdash e f : \mathsf{C}\tau}$$

$$\frac{\Gamma \vdash e : [\mathsf{bool}] \quad \Gamma \vdash f : \mathsf{C}\tau \quad \Gamma \vdash g : \mathsf{C}\tau}{\Gamma \vdash \mathsf{if}\, e\, \mathsf{then}\, f\, \mathsf{else}\, g : \mathsf{C}\tau} \quad \frac{}{\Gamma \vdash \delta : \mathsf{C}\tau} \quad \frac{\Gamma \vdash e : \mathsf{C}\tau \Gamma \vdash f : \mathsf{C}\tau}{\Gamma \vdash e \,\Box\, f : \mathsf{C}\tau} \quad \frac{\Gamma, x : \mathsf{C}\tau \vdash e : \mathsf{C}\tau}{\Gamma \vdash \mathsf{fix}(x = e) : \mathsf{C}\tau}$$

$$\frac{\Gamma \vdash e : [\mathsf{chan}], f : \tau}{\Gamma \vdash e!_\tau f : \mathsf{C}I} \quad \frac{\Gamma \vdash e : [\mathsf{chan}]}{\Gamma \vdash e?_\tau : \mathsf{C}\tau} \quad \frac{\Gamma \vdash e : \mathsf{C}\sigma, f : \mathsf{C}\tau}{\Gamma \vdash e \,\|\, f : \mathsf{C}\tau} \quad \frac{\Gamma \vdash e : \mathsf{C}\tau, f_1 : [\mathsf{chan}],\ldots,f_n : [\mathsf{chan}]}{\Gamma \vdash e \upharpoonright f_1,\ldots,f_n : \mathsf{C}\tau}$$

Table 1: Typing rules for $\mathsf{CMML}\Sigma$

- an *internal transition* relation $e \xrightarrow{\text{´}} e'$,
- a *termination* relation $e \xrightarrow{\surd g} e'$,
- an *output* relation $e \xrightarrow{f!_\sigma g} e'$,
- an *input* relation $e \xrightarrow{f?_\sigma x} e'$, and
- a *deadlocked* term $\delta$,

with the typing:

- if $e \xrightarrow{\text{´}} e'$ then $\vdash e : \mathsf{C}\tau$ and $\vdash e' : \mathsf{C}\tau$,
- if $e \xrightarrow{\surd f} e'$ then $\vdash e : \mathsf{C}\tau$, $\vdash f : \tau$, and $\vdash e' : \mathsf{C}\tau$,
- if $e \xrightarrow{f!_\sigma g} e'$ then $\vdash e : \mathsf{C}\tau$, $\vdash f : [\mathsf{chan}]$, $\vdash g : \sigma$, and $\vdash e' : \mathsf{C}\tau$, and
- if $e \xrightarrow{f?_\sigma x} e'$ then $\vdash e : \mathsf{C}\tau$, $\vdash f : [\mathsf{chan}]$, and $x : \sigma \vdash e' : \mathsf{C}\tau$,

where $\delta$ has no reductions, and where $\xrightarrow{\surd f}$ has the properties:

- if $e \xrightarrow{\surd f} e'$ then $e' \not\xrightarrow{\surd g}$,
- if $e \xrightarrow{\surd f} \xmapsto{\mu} e'$ then $e \xmapsto{\mu} \xrightarrow{\surd f} e'$, and
- if $e \xrightarrow{\surd f} e'$ and $e \xmapsto{\mu} e''$ are distinct transitions, then $e' \xmapsto{\mu} e'''$ and $e'' \xrightarrow{\surd f} e'''$ for some $e'''$,

where we write $e \xmapsto{\mu} e'$ for the *early* operational semantics given by:

- $e \xmapsto{\text{´}} e'$ iff $e \xrightarrow{\text{´}} e'$,
- $e \xmapsto{\surd f} e'$ iff $e \xrightarrow{\surd f} e'$,
- $e \xmapsto{f!_\tau f'} e'$ iff $e \xrightarrow{f!_\tau f'} e'$, and
- $e \xmapsto{f?_\tau f'} e'[f'/x]$ iff $e \xrightarrow{f?_\tau x} e'$.

Let $a$ range over visible actions $e!_\tau f$ and $e?_\tau x$, let $\alpha$ range over $a$ and ´ , and let $\mu$ range over all actions.

We can de£ne an operational semantics for terms of the form $d\vec{e}$ or $\delta$, such that the reductions of $\mathsf{eq}$ are (when $e \neq f$):

$$\mathsf{eq}(e,e) \xrightarrow{\surd \mathsf{true}} \delta \qquad \mathsf{eq}(e,f) \xrightarrow{\surd \mathsf{false}} \delta$$

For example, the operational semantics for $\mathsf{NatList}$ is:

$$\begin{array}{cc}
\mathsf{eq}(e,e) \xrightarrow{\surd \mathsf{true}} \delta & \mathsf{eq}(e,f) \xrightarrow{\surd \mathsf{false}} \delta \\
\mathsf{isnil}(\mathsf{nil}) \xrightarrow{\surd \mathsf{true}} \delta & \mathsf{isnil}(\mathsf{cons}(e,f)) \xrightarrow{\surd \mathsf{false}} \delta \\
\mathsf{hd}(\mathsf{cons}(e,f)) \xrightarrow{\surd e} \delta & \mathsf{tl}(\mathsf{cons}(e,f)) \xrightarrow{\surd f} \delta
\end{array}$$
$$\mathsf{pred}(\mathsf{succ}\, e) \xrightarrow{\surd e} \delta$$

Note that we have not given any reductions for $\mathsf{pred}\,\mathsf{zero}$, $\mathsf{hd}\,\mathsf{nil}$ or $\mathsf{tl}\,\mathsf{nil}$, and so they deadlock.

Given a higher-order symbolic vps for terms of the form $d\vec{e}$ and $\delta$, we can extend it to $\mathsf{CMML}\Sigma$ as in Table 2.

Write $e \overset{\mu}{\Longrightarrow} e'$ for $e \Longrightarrow\xrightarrow{\mu} e'$, and $e \overset{\hat{\mu}}{\Longrightarrow} e'$ for $e \overset{\mu}{\Longrightarrow} e'$ or $e = e'$ and $\mu =$ ´ .

For example, one reduction of the buffer $\mathsf{buff}(i,o,\mathsf{nil})$ is:

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{buff}(i,o,\mathsf{nil})} \\
\Longrightarrow & \quad \mathsf{let}\, w \Leftarrow i?_{\mathsf{nat}}\, \mathsf{in} \\
 & \quad \mathsf{let}\, z \Leftarrow \mathsf{snoc}(\mathsf{nil},w)\, \mathsf{in} \\
 & \quad \mathsf{buff}(i,o,z) \\
 & \Box\, \mathsf{let}\, w \Leftarrow \mathsf{hd}\,\mathsf{nil}\, \mathsf{in} \\
 & \quad \mathsf{let}\, v \Leftarrow o!_{\mathsf{nat}} w\, \mathsf{in} \\
 & \quad \mathsf{let}\, z \Leftarrow \mathsf{tl}\,\mathsf{nil}\, \mathsf{in} \\
 & \quad \mathsf{buff}(i,o,z) \\
\xmapsto{i?n} & \mathsf{let}\, w \Leftarrow [n]\, \mathsf{in} \\
 & \mathsf{let}\, z \Leftarrow \mathsf{snoc}(\mathsf{nil},w)\, \mathsf{in} \\
 & \mathsf{buff}(i,o,z) \\
\xrightarrow{\text{´}} & \mathsf{let}\, z \Leftarrow \mathsf{snoc}(\mathsf{nil},n)\, \mathsf{in} \\
 & \mathsf{buff}(i,o,z) \\
\Longrightarrow & \mathsf{buff}(i,o,\mathsf{cons}(n,\mathsf{nil}))
\end{array}$$

$$
\frac{}{e!_\tau f \xrightarrow{e!_\tau f} [*]} \quad \frac{}{e?_\tau \xrightarrow{e?_\tau x} [x]} \quad \frac{}{[e] \xrightarrow{\surd e} \delta}
$$

$$
\frac{e \xrightarrow{\alpha} e'}{\mathsf{let}\,x \Leftarrow e\,\mathsf{in}\,f \xrightarrow{\alpha} \mathsf{let}\,x \Leftarrow e'\,\mathsf{in}\,f} \quad \frac{e \xrightarrow{\surd g} e'}{\mathsf{let}\,x \Leftarrow e\,\mathsf{in}\,f \overset{\acute{}}{\longrightarrow} e' \parallel f[g/x]} \quad \frac{}{(\lambda x\,.\,e)\,f \overset{\acute{}}{\longrightarrow} e[f/x]}
$$

$$
\frac{}{\mathsf{fix}(x = e) \overset{\acute{}}{\longrightarrow} e[\mathsf{fix}(x = e)/x]} \quad \frac{}{\mathsf{if}\,\mathsf{true}()\,\mathsf{then}\,f\,\mathsf{else}\,g \overset{\acute{}}{\longrightarrow} f} \quad \frac{}{\mathsf{if}\,\mathsf{false}()\,\mathsf{then}\,f\,\mathsf{else}\,g \overset{\acute{}}{\longrightarrow} g}
$$

$$
\frac{e \overset{\acute{}}{\longrightarrow} e'}{e \,\square\, f \overset{\acute{}}{\longrightarrow} e' \,\square\, f} \quad \frac{e \xrightarrow{a} e'}{e \,\square\, f \xrightarrow{a} e'} \quad \frac{e \xrightarrow{\surd g} e'}{e \,\square\, f \overset{\acute{}}{\longrightarrow} e' \parallel [g]} \quad \frac{f \overset{\acute{}}{\longrightarrow} f'}{e \,\square\, f \overset{\acute{}}{\longrightarrow} e \,\square\, f'} \quad \frac{f \xrightarrow{a} f'}{e \,\square\, f \xrightarrow{a} f'} \quad \frac{f \xrightarrow{\surd g} f'}{e \,\square\, f \overset{\acute{}}{\longrightarrow} f' \parallel [g]}
$$

$$
\frac{e \xrightarrow{\alpha} e'}{e \parallel f \xrightarrow{\alpha} e' \parallel f} \quad \frac{e \xrightarrow{g!_\tau h} e' \quad f \xrightarrow{g?_\tau x} f'}{e \parallel f \overset{\acute{}}{\longrightarrow} e' \parallel f'[h/x]} \quad \frac{f \xrightarrow{\mu} f'}{e \parallel f \xrightarrow{\mu} e \parallel f'} \quad \frac{e \xrightarrow{g?_\tau x} e' \quad f \xrightarrow{g!_\tau h} f'}{e \parallel f \overset{\acute{}}{\longrightarrow} e'[h/x] \parallel f'}
$$

$$
\frac{e \overset{\acute{}}{\longrightarrow} e'}{e \restriction \vec{f} \overset{\acute{}}{\longrightarrow} e' \restriction \vec{f}} \quad \frac{e \xrightarrow{\surd g} e'}{e \restriction \vec{f} \xrightarrow{\surd g} \restriction \vec{f}} \quad \frac{e \xrightarrow{f?x} e'}{e \restriction \vec{f}, f, \vec{f}' \xrightarrow{f?_\tau x} e' \restriction \vec{f}, f, \vec{f}'} \quad \frac{e \xrightarrow{f!g} e'}{e \restriction \vec{f}, f, \vec{f}' \xrightarrow{f!_\tau g} e' \restriction \vec{f}, f, \vec{f}'}
$$

Table 2: Operational semantics for $\mathsf{CMML}\,\Sigma$

$$
\begin{array}{ll}
\Longrightarrow & \mathsf{let}\,w \Leftarrow i?_{\mathsf{nat}}\,\mathsf{in} \\
& \mathsf{let}\,z \Leftarrow \mathsf{snoc}(\mathsf{cons}(n, \mathsf{nil}), w)\,\mathsf{in} \\
& \mathsf{buff}(i, o, z) \\
& \square\,\mathsf{let}\,w \Leftarrow \mathsf{hd}(\mathsf{cons}(n, \mathsf{nil}))\,\mathsf{in} \\
& \mathsf{let}\,v \Leftarrow o!_{\mathsf{nat}} w\,\mathsf{in} \\
& \mathsf{let}\,z \Leftarrow \mathsf{tl}(\mathsf{cons}(n, \mathsf{nil}))\,\mathsf{in} \\
& \mathsf{buff}(i, o, z) \\
\overset{\acute{}}{\longrightarrow} & \mathsf{let}\,w \Leftarrow i?_{\mathsf{nat}}\,\mathsf{in} \\
& \mathsf{let}\,z \Leftarrow \mathsf{snoc}(\mathsf{cons}(n, \mathsf{nil}), w)\,\mathsf{in} \\
& \mathsf{buff}(i, o, z) \\
& \square\,\mathsf{let}\,v \Leftarrow o!_{\mathsf{nat}} n\,\mathsf{in} \\
& \mathsf{let}\,z \Leftarrow \mathsf{tl}(\mathsf{cons}(n, \mathsf{nil}))\,\mathsf{in} \\
& \mathsf{buff}(i, o, z) \\
\xrightarrow{o!n} & \mathsf{let}\,v \Leftarrow [*]\,\mathsf{in} \\
& \mathsf{let}\,z \Leftarrow \mathsf{tl}(\mathsf{cons}(n, \mathsf{nil}))\,\mathsf{in} \\
& \mathsf{buff}(i, o, z) \\
\overset{\acute{}}{\longrightarrow} & \mathsf{let}\,z \Leftarrow \mathsf{tl}(\mathsf{cons}(n, \mathsf{nil}))\,\mathsf{in} \\
& \mathsf{buff}(i, o, z) \\
\overset{\acute{}}{\longrightarrow} & \mathsf{buff}(i, o, \mathsf{nil})
\end{array}
$$

De£ne may-testing for $\mathsf{CMML}\,\Sigma$ as $\Gamma \models e \sqsubseteq_O f : \tau$ iff $C[e] \xRightarrow{\surd} *$ implies $C[f] \xRightarrow{\surd *}$ for any closing context $C$ of type $\mathsf{C}I$.

## 4 Biimulation

We can de£ne a notion of higher-order late bisimulation for $\mathsf{CMML}\,\Sigma$, based on Abramsky's [2] applicative simulation for the untyped $\lambda$-calculus, Milner, Parrow and Walker's [20] late bisimulation for the $\pi$-calculus, and Howe's [13] simulation for lazy computation.

A (higher-order late) simulation on $\mathsf{CMML}\,\Sigma$ is a type-indexed family of relations $\mathcal{R}_\tau \subseteq \{(e, f) \mid \vdash e, f : \tau\}$ such that:

- if $e \mathrel{\mathcal{R}_{[A]}} f$ then $e = f$.
- if $(e, e') \mathrel{\mathcal{R}_{\sigma \otimes \tau}} (f, f')$ then $e \mathrel{\mathcal{R}_\sigma} f$ and $e' \mathrel{\mathcal{R}_\tau} f'$,
- if $(\lambda x\,.\,e) \mathrel{\mathcal{R}_{\sigma \to \mathsf{C}\tau}} (\lambda y\,.\,f)$ then for all $\vdash g : \sigma$ we have $e[g/x] \mathrel{\mathcal{R}_{\mathsf{C}\tau}} f[g/y]$,
- if $e \mathrel{\mathcal{R}_{\mathsf{C}\tau}} f$ and $e \overset{\acute{}}{\longrightarrow} e'$ then $f \Longrightarrow f'$ and $e' \mathrel{\mathcal{R}_{\mathsf{C}\tau}} f'$,
- if $e \mathrel{\mathcal{R}_{\mathsf{C}\tau}} f$ and $e \xrightarrow{\surd e'} e''$ then $f \xRightarrow{\surd f'} f''$ and $(e', e'') \mathrel{\mathcal{R}_{\tau \otimes \mathsf{C}\tau}} (f', f'')$,
- if $e \mathrel{\mathcal{R}_{\mathsf{C}\tau}} f$ and $e \xrightarrow{g!_\sigma e'} e''$ then $f \xRightarrow{g!_\sigma f'} f''$ and $(e', e'') \mathrel{\mathcal{R}_{\sigma \otimes \mathsf{C}\tau}} (f', f'')$, and
- if $e \mathrel{\mathcal{R}_{\mathsf{C}\tau}} f$ and $e \xrightarrow{g?_\sigma x} e'$ then $f \xRightarrow{g?_\sigma y} f'$ and $(\lambda x\,.\,e') \mathrel{\mathcal{R}_{\sigma \to \mathsf{C}\tau}} (\lambda y\,.\,f')$.

A *bisimulation* is a simulation whose inverse is also a simulation. Then de£ne:

- *simulation preorder* $\preceq$ is the largest simulation,
- *mutual simulation equivalence* $\asymp$ is $\preceq \cap \succeq$.
- *bisimulation equivalence* $\approx$ is the largest bisimulation.

Given a type-indexed relation $\mathcal{R}_\tau$ of closed terms, let $\mathcal{R}^\circ_{\Gamma,\tau}$ be the corresponding relation on open terms:

$$
\mathcal{R}^\circ_{\vec{x}:\vec{\sigma},\tau} = \{(e, f) \mid \forall \vdash \vec{g} : \vec{\sigma}\,.\,e[\vec{g}/\vec{x}] = f[\vec{g}/\vec{x}]\}
$$

We shall often elide the indexes from these relations, writing $e \mathrel{\mathcal{R}} f$ rather than $e \mathrel{\mathcal{R}_\tau} f$ and $e \mathrel{\mathcal{R}^\circ} f$ for $e \mathrel{\mathcal{R}^\circ_{\Gamma,\tau}} f$ when context makes the typing obvious.

Note that bisimulation is strictly £ner then mutual simulation, for example:

$$
\begin{array}{rcl}
a? \,\square\, \mathsf{let}\,x \Leftarrow a?\,\mathsf{in}\,\delta & \asymp & a? \\
a? \,\square\, \mathsf{let}\,x \Leftarrow a?\,\mathsf{in}\,\delta & \not\approx & a?
\end{array}
$$

As this example shows, mutual simulation does not have the power to detect deadlock, which is why Milner [19, exercise 14] chose to use bisimulation rather than mutual simulation for CCS.

In earlier versions of this paper, it was necessary to use mutual simulation rather than bisimulation, because I was unable to £nd a proof that bisimulation was a congruence for

$\mathsf{CMML}\,\Sigma$. However, Andy Pitts has since shown me an unpublished proof of Howe's [12] which can be adapted to show that bisimulation is a congruence for $\mathsf{CMML}\,\Sigma$.

The full paper uses Gordon's variant [8] of Howe's proof technique [13, 12] to show that bisimulation is a congruence for $\mathsf{CMML}\,\Sigma$.

Let $\mathsf{CMML}\,\Sigma$ be the signature with types as sorts, and judgements $\vec{x} : \vec{\sigma} \vdash e : \tau$ as constructors $\vec{\sigma} \to \tau$, viewed up to bisimulation. Any signature morphism $f : \Sigma \to \Sigma'$ extends homomorphically to a signature morphism $\mathsf{CMML}\,f : \mathsf{CMML}\,\Sigma \to \mathsf{CMML}\,\Sigma'$, and it is routine to show that $\mathsf{CMML} : \mathbf{SigBCD} \to \mathbf{SigBCD}$ is a functor.

The equations in Table 3 include Moggi's equations for the monadic metalanguage, and so we can show that $\mathsf{CMML}\,\Sigma$ has categorical structure given by Table 4.

**Proposition 1** $\mathsf{CMML}\,\Sigma$ *forms a category with £nite products, a monad $T : \mathsf{CMML}\,\Sigma \to \mathsf{CMML}\,\Sigma$, and all $T$-exponentials.*

**Proof** The equations in Table 3 are suf£cient to show that the structure de£ned in Table 4 has the required properties. This is shown in [22] and the full version of this paper. □

## 5 Denotational semantics

Let $\mathbf{Alg}$ be the category of algebraic dcpo's, together with continuous morphisms (we are not requiring dcpo's to have least elements). Let $\mathbf{Alg}_{\perp\vee}$ be the category of algebraic dcpo's with all £nite joins, together with continuous morphisms which respect the joins. Let $X \to Y$ be the continuous function space between $X$ and $Y$, and let $X \to_\vee Y$ be the continuous $\vee$-respecting function space between $X$ and $Y$.

For any indexing sets $J$ and $K$ and for any object $(\vec{I}, \vec{O}, V)$ in $\mathbf{Alg}^{\mathrm{op}\,J} \times \mathbf{Alg}^K \times \mathbf{Alg}$, a *pre-process domain* over $(\vec{I}, \vec{O}, V)$ is an object $P$ in $\mathbf{Alg}_{\perp\vee}$ with morphisms:

$$\begin{aligned} \mathsf{in}_j\, c &: (I_j \to P) \to_\vee P \\ \mathsf{out}_k\, c &: O_k \to P \to_\vee P \\ \mathsf{val} &: V \to P \to_\vee P \end{aligned}$$

for each $c \in [\![\mathsf{chan}]\!]$, $j \in J$ and $k \in K$. A pre-process domain morphism is an $\mathbf{Alg}_{\perp\vee}$ morphism which respects $\mathsf{in}_j\, c$, $\mathsf{out}_k\, c$ and $\mathsf{val}$. A *process domain* is a pre-process domain where:

$$\begin{aligned} \mathsf{in}_j\, c(f; \mathsf{val}\, v) &\leq \mathsf{val}\, v(\mathsf{in}_j\, c f) \\ \mathsf{out}_k\, cd(\mathsf{val}\, vp) &\leq \mathsf{val}\, v(\mathsf{out}_k\, cdp) \\ \mathsf{val}\, v(\mathsf{val}\, wp) &= \mathsf{val}\, vp \end{aligned} \tag{1}$$

Let $\mathsf{Proc}(\vec{I}, \vec{O}, V)$ be the initial process domain over $(\vec{I}, \vec{O}, V)$.

**Proposition 2** $\mathsf{Proc}$ *is a continuous and locally continuous functor* $\mathbf{Alg} \to \mathbf{Alg}$.

**Proof** Given in the full version of this paper. □

Hennessy [10] has proposed a domain for higher-order processes which is the canonical £xed point of the functor:

$$X \mapsto \prod_c (X \to X)_\perp \times \prod_c (X \otimes_r X)$$

where $\otimes_r$ is the left adjoint to $\to_\vee$:

$$\mathsf{curry}_r : \mathbf{Alg}_\vee[X \otimes_r Y, Z] \simeq \mathbf{Alg}[X, Y \to_\vee Z]$$

We can extend this to typed processes by de£ning $\mathsf{PreProc}(\vec{I}, \vec{O}, V)$ to be the canonical £xed point of the functor:

$$X \mapsto \prod_{j,c} (I_j \to X)_\perp \times \prod_{k,c} (O_k \otimes_r X)_\perp \times (V \otimes X)_\perp$$

**Proposition 3**

1. *$\mathsf{PreProc}(\vec{I}, \vec{O}, V)$ is the initial pre-process domain.*
2. *$\mathsf{Proc}(\vec{I}, \vec{O}, V)$ is $\mathsf{PreProc}(\vec{I}, \vec{O}, V)$ quotiented by the process domain preorder (1).*

**Proof** Given in the full version of this paper. □

De£ne the *traces* $\mathsf{Trace}(\vec{I}, \vec{O}, V) \subseteq \mathsf{Proc}(\vec{I}, \vec{O}, V)$ as the elements given by the grammar:

$$\begin{aligned} s &::= \perp \mid \mathsf{in}_j\, c(d \Rightarrow s) \mid \mathsf{out}_k\, cds \mid \mathsf{val}\, vu \\ u &::= \perp \mid \mathsf{in}_j\, c(d \Rightarrow u) \mid \mathsf{out}_k\, cdu \end{aligned}$$

for compact $d$ and $v$.

**Proposition 4** *$p$ is compact iff $p = s_1 \vee \cdots \vee s_n$ for traces $s_i$.*

**Proof** 'If' follows from showing by induction on $s$ that $s$ is compact. 'Only if' follows by showing that any $p$ is the join of all the traces below it. □

Any continuous function is determined by its effect on compact elements, and so we can de£ne functions $\mathsf{Proc}(\vec{I}, \vec{O}, V) \to_{\perp\vee} D$ by giving their effect on traces. For example, the restriction operator:

$$\_ \restriction \_ : \mathsf{Proc}(\vec{I}, \vec{O}, V) \to_{\perp\vee} [\![\mathsf{chan}]\!]^n \to \mathsf{Proc}(\vec{I}, \vec{O}, V)$$

is de£ned by its action on traces:

$$\begin{aligned} (\mathsf{in}_j\, c(d \Rightarrow s)) \restriction \vec{c} &= \begin{cases} \mathsf{in}_j\, c(d \Rightarrow (s \restriction \vec{c})) & \text{if } c \in \vec{c} \\ \perp & \text{otherwise} \end{cases} \\ (\mathsf{out}_k\, cds) \restriction \vec{c} &= \begin{cases} (\mathsf{out}_k\, cd(s \restriction \vec{c})) & \text{if } c \in \vec{c} \\ \perp & \text{otherwise} \end{cases} \\ (\mathsf{val}\, vu) \restriction \vec{c} &= \mathsf{val}\, v(u \restriction \vec{c}) \end{aligned}$$

We can de£ne concurrency, and the monad natural transformations similarly:

$$\begin{aligned} \_ \| \_ &: \mathsf{Proc}(\vec{I}, \vec{O}, V) \to_\vee \mathsf{Proc}(\vec{I}, \vec{O}, W) \to_\vee \mathsf{Proc}(\vec{I}, \vec{O}, W) \\ \_ * \_ &: (V \to W) \to \mathsf{Proc}(\vec{I}, \vec{O}, V) \to_{\perp\vee} \mathsf{Proc}(\vec{I}, \vec{O}, W) \\ \mu &: \mathsf{Proc}(\vec{D}, \vec{D}, \mathsf{Proc}(\vec{D}, \vec{D}, V)) \to_{\perp\vee} \mathsf{Proc}(\vec{D}, \vec{D}, V) \\ \eta &: V \to \mathsf{Proc}(\vec{I}, \vec{O}, V) \end{aligned}$$

$$\begin{array}{rcl}
x & \approx^\circ & * \\
(v.\mathrm{L}, v.\mathrm{R}) & \approx^\circ & v \\
\mathsf{let}\, x \Leftarrow [e]\, \mathsf{in}\, f & \approx^\circ & f[e/x] \\
\mathsf{let}\, x \Leftarrow e\, \mathsf{in}\, [x] & \approx^\circ & e \\
\mathsf{let}\, y \Leftarrow (\mathsf{let}\, x \Leftarrow e\, \mathsf{in}\, f)\, \mathsf{in}\, g & \approx^\circ & \mathsf{let}\, x \Leftarrow e\, \mathsf{in}\, (\mathsf{let}\, y \Leftarrow f\, \mathsf{in}\, g) \\
(\lambda x \,.\, e) f & \approx^\circ & e[f/x] \\
\lambda y \,.\, (g y) & \approx^\circ & g
\end{array}$$

Table 3: Some bisimulations for $\mathsf{CMML}\,\Sigma$ ($y$ not free in $g$)

$$\begin{array}{rcl}
\mathsf{id}_\tau & = & (x : \tau \vdash x : \tau) \\
(x : \rho \vdash e : \sigma); (y : \sigma \vdash f : \tau) & = & (x : \rho \vdash f[e/x] : \tau) \\
1 & = & I \\
!_\tau & = & (x : \tau \vdash * : I) \\
\sigma \times \tau & = & \sigma \otimes \tau \\
\pi & = & (x : \sigma \otimes \tau \vdash x.\mathrm{L} : \sigma) \\
\pi' & = & (x : \sigma \otimes \tau \vdash x.\mathrm{R} : \tau) \\
T\tau & = & \mathsf{C}\,\tau \\
T(x : \sigma \vdash e : \tau) & = & (y : \mathsf{C}\,\sigma \vdash \mathsf{let}\, x = y\, \mathsf{in}\, [e] : \mathsf{C}\,\tau) \\
\eta_\tau & = & (x : \tau \vdash [x] : \mathsf{C}\,\tau) \\
\mu_\tau & = & (x : \mathsf{C}\,\mathsf{C}\,\tau \vdash \mathsf{let}\, y \Leftarrow x\, \mathsf{in}\, y : \mathsf{C}\,\tau) \\
t_{\sigma,\tau} & = & (x : \sigma \otimes \mathsf{C}\,\tau \vdash \mathsf{let}\, y \Leftarrow x.\mathrm{R}\, \mathsf{in}\, [(x.\mathrm{L}, y)] : \mathsf{C}(\sigma \otimes \tau)) \\
T\tau^\sigma & = & \sigma \to \mathsf{C}\,\tau \\
\mathsf{curry}(x : \rho \otimes \sigma \vdash e : \mathsf{C}\,\tau) & = & (y : \rho \vdash \lambda z \,.\, \mathsf{let}\, x \Leftarrow [(y, z)]\, \mathsf{in}\, e : \sigma \to \mathsf{C}\,\tau) \\
\mathsf{curry}^{-1}(x : \rho \vdash e : \sigma \to \mathsf{C}\,\tau) & = & (y : \rho \otimes \sigma \vdash e(y.\mathrm{R}) : \mathsf{C}\,\tau)
\end{array}$$

Table 4: Categorical structure of $\mathsf{CMML}\,\Sigma$

**Proposition 5** $\mathsf{Proc}(\vec{D}, \vec{D}, \_) : \mathbf{Alg} \to \mathbf{Alg}$ *is a monad.*

**Proof** Given in the full version of this paper. $\square$

Given a semantics $[\![\_]\!] : \Sigma \to \mathbf{Alg}$ for $\Sigma$, we extend it to $\mathsf{CMML}\,\Sigma$ by giving the an object $[\![\tau]\!]$ in $\mathbf{Alg}$ for each type $\tau$:

$$\begin{array}{rcl}
[\![I]\!] & = & 1 \\
[\![\sigma \otimes \tau]\!] & = & [\![\sigma]\!] \times [\![\tau]\!] \\
[\![\sigma \to \mathsf{C}\,\tau]\!] & = & [\![\sigma]\!] \to [\![\mathsf{C}\,\tau]\!] \\
[\![\mathsf{C}\,\tau]\!] & \simeq & \mathsf{Proc}(\langle [\![\sigma]\!] \mid \sigma \in \mathbf{T} \rangle, \langle [\![\sigma]\!] \mid \sigma \in \mathbf{T} \rangle, [\![\tau]\!])
\end{array}$$

where $\mathbf{T}$ is the set of all CMML types, and for each $\vec{x} : \vec{\sigma} \vdash e : \tau$ a morphism:

$$[\![x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash e : \tau]\!] : [\![\sigma_1]\!] \times \cdots \times [\![\sigma_n]\!] \to [\![\tau]\!]$$

given by:

- $[\![*]\!]$ and $[\![(e, f)]\!]$ are given by the products in $\mathbf{Alg}$,
- $[\![[e]]\!]$ and $[\![\mathsf{let}\, x \Leftarrow e\, \mathsf{in}\, f]\!]$ are given by the monadic structure of $\mathsf{Proc}$,
- $[\![\lambda x \,.\, e]\!]$ and $[\![e f]\!]$ are given by the $\mathsf{Proc}$-exponentials in $\mathbf{Alg}$,
- $[\![\mathsf{if}\, e\, \mathsf{then}\, f\, \mathsf{else}\, g]\!]$ is given by the coproducts in $\mathbf{Alg}$,
- $[\![\delta]\!]$ and $[\![e \,\Box\, f]\!]$ are given by bottom and join in $\mathsf{Proc}[\![\tau]\!]$,
- $[\![\mathsf{fix}(x = e)]\!]$ is the least £xed point of $x \mapsto [\![e]\!]$, and
- $[\![e?]\!]$, $[\![e!\,f]\!]$, $[\![e \parallel f]\!]$ and $[\![e \restriction \vec{f}]\!]$ are given as above.

This semantics is de£ned in full in the full version of the paper.

We shall sometimes elide the type information, and write $[\![e]\!]$ for $[\![\Gamma \vdash e : \tau]\!]$ where this is unambiguous.

A semantics $[\![\_]\!] : \Sigma \to \mathbf{Alg}$ is *adequate* iff:

$$[\![\vdash d\vec{e} : \mathsf{C}[A]]\!] = \bigvee \{ [\![\vdash [f] : \mathsf{C}[A]]\!] \mid d\vec{e} \overset{\surd}{\Longrightarrow} f \}$$

A semantics $[\![\_]\!] : \Sigma \to \mathbf{Alg}$ is *expressive* iff for any compact $a \in [\![A]\!]$ we can £nd terms $\mathsf{is}_a$ and $\mathsf{test}_a$ such that:

$$[\![\vdash \mathsf{is}_a : [A]]\!] = a \qquad [\![\vdash \mathsf{test}_a : [A] \to \mathsf{C}\, I]\!] = (a \Rightarrow \eta\bot)$$

A semantics $[\![\_]\!] : \mathsf{CMML}\,\Sigma \to \mathbf{Alg}$ is *correct* iff:

$$[\![\Gamma \vdash e : \tau]\!] \le [\![\Gamma \vdash f : \tau]\!] \text{ implies } \Gamma \models e \sqsubseteq_O f : \tau$$

The semantics for $\mathsf{CMML}\,\Sigma$ is *fully abstract* iff:

$$[\![\Gamma \vdash e : \tau]\!] \le [\![\Gamma \vdash f : \tau]\!] \text{ iff } \Gamma \models e \sqsubseteq_O f : \tau$$

We will now sketch the proof that if a semantics for $\Sigma$ is adequate then its extension to $\mathsf{CMML}\,\Sigma$ is correct, and that if a semantics for $\Sigma$ is adequate and expressive, then its extension to $\mathsf{CMML}\,\Sigma$ is fully abstract.

## 6 Program logic

In order to show the relationship between the operational and denotational semantics of $\mathsf{CMML}\,\Sigma$, we shall use a *program logic* similar to that used by Abramsky [2] and Ong [23] in modelling the untyped $\lambda$-calculus, based on Abramsky's [3] *domain theory in logical form*.

The *program logic* for $\mathsf{CMML}\,\Sigma$ has propositions:

$$\phi ::= * \mid (\phi,\psi) \mid \lfloor a \rfloor \mid \omega \mid \phi \wedge \psi \mid \phi \Rightarrow \psi \mid \langle c?_\sigma\rangle\phi \mid \langle c!_\sigma\rangle\phi \mid \langle\surd\rangle\phi$$

These can be statically typed, so the propositions for type $\tau$ are those where $\phi : \mathcal{L}\tau$, given by the type system in Table 5.

The operational characterization of the logic has judgements $\models e : \phi$ given for closed terms by Table 6. This can be generalized to open terms as:

$$\vec{x} : \vec{\phi} \models e : \psi \text{ iff } \forall \models \vec{f} : \vec{\phi} \,.\, \models e[\vec{f}/\vec{x}] : \psi$$

Let $\Delta$ range over propositional contexts of the form $x_1 : \phi_1, \ldots, x_n : \phi_n$, and write $\Delta : \mathcal{L}\Gamma$ for:

$$\begin{array}{l}(x_1 : \phi_1, \ldots, x_n : \phi_n) : \mathcal{L}(x_1 : \tau_1, \ldots, x_n : \tau_n) \\ \quad \text{iff } \phi_1 : \mathcal{L}\tau_1, \ldots, \phi_n : \mathcal{L}\tau_n\end{array}$$

We can also de£ne a denotational semantics for propositions, so that if $\phi : \mathcal{L}\tau$ then $[\![\phi]\!] \in [\![\tau]\!]$:

$$\begin{array}{rclrcl}
[\![*]\!] &=& \bot & [\![(\phi,\psi)]\!] &=& ([\![\phi]\!],[\![\psi]\!]) \\
[\![\omega]\!] &=& \bot & [\![\phi\wedge\psi]\!] &=& [\![\phi]\!] \vee [\![\psi]\!] \\
[\![\lfloor a \rfloor]\!] &=& a & [\![\phi\Rightarrow\psi]\!] &=& [\![\phi]\!] \Rightarrow [\![\psi]\!] \\
[\![\langle c?_\sigma\rangle\phi]\!] &=& \mathsf{in}\,c[\![\phi]\!] & [\![\langle c!_\sigma\rangle(\phi,\psi)]\!] &=& \mathsf{out}\,c[\![\phi]\!][\![\psi]\!] \\
[\![\langle\surd\rangle(\phi,\psi)]\!] &=& \mathsf{val}[\![\phi]\!][\![\psi]\!] &&&
\end{array}$$

Whenever $\Delta : \mathcal{L}\Gamma$, we can de£ne $[\![\Delta]\!] \in [\![\Gamma]\!]$ as:

$$[\![x_1 : \phi_1, \ldots, x_n : \phi_n]\!] = ([\![\phi_1]\!], \ldots, [\![\phi_n]\!])$$

**Proposition 6** $a \in [\![\tau]\!]$ *is compact iff* $\exists \phi : \mathcal{L}\tau \,.\, a = [\![\phi]\!]$.

**Proof** 'If' is an induction on $\phi$. 'Only if' relies on Proposition 4. □

## 7 Full abstraction

We can now show that the semantics for $\mathsf{CMML}\,\Sigma$ is fully abstract. We begin by showing that if $\Sigma$ is expressive, then so is $\mathsf{CMML}\,\Sigma$.

The *primes* of the logic are given by:

$$\begin{array}{rcl}
\pi &::=& \omega \mid \langle c?_\tau\rangle(\phi\Rightarrow\pi) \mid \langle c!_\tau\rangle(\phi,\pi) \mid \langle\surd\rangle(\phi,\nu) \\
\nu &::=& \omega \mid \langle c?_\tau\rangle(\phi\Rightarrow\nu) \mid \langle c!_\tau\rangle(\phi,\nu)
\end{array}$$

Note that $[\![\pi]\!]$ is a trace, and so by Proposition 4 for any $\phi \in \mathcal{L}(\mathsf{C}\,\tau)$ we can £nd $\pi_i$ such that $\phi = \pi_1 \wedge \cdots \wedge \pi_n$.

**Proposition 7** *If the semantics for* $\Sigma$ *is expressive, then for any* $\phi : \mathcal{L}\tau$ *we can £nd a term* $\vdash \mathsf{term}\,\phi : \tau$ *such that* $[\![\phi]\!] = [\![\mathsf{term}\,\phi]\!]$.

**Proof** Let $\mathsf{term}_\tau\,\phi$ be de£ned as in Table 7, where $\mathsf{result}$ is an unused channel, and when $\pi \in \mathcal{L}(\mathsf{C}\,\tau)$ then $\overline{\pi} \in \mathcal{L}(\mathsf{C}\,I)$ is:

$$\begin{array}{rcl}
\overline{\omega} &=& [*] \\
\overline{\langle c?_\sigma\rangle(\phi\Rightarrow\pi)} &=& \langle c!_\sigma\rangle(\phi,\overline{\pi}) \\
\overline{\langle c!_\sigma\rangle(\phi,\pi)} &=& \langle c?_\sigma\rangle(\phi\Rightarrow\overline{\pi}) \\
\overline{\langle\surd\rangle(\phi,\nu)} &=& \langle \mathsf{result}?_\tau\rangle(\phi\Rightarrow\overline{\nu})
\end{array}$$

Then show by induction on $\phi$ that $[\![\mathsf{term}\,\phi]\!] = [\![\phi]\!]$. □

We can then verify that: $[\![\phi]\!] = [\![\vdash \mathsf{term}_\tau\,\phi : \tau]\!]$ This expressivity result is used in showing that the semantics for $\mathsf{CMML}\,\Sigma$ is fully abstract. The relationship between expressivity and full abstraction has been long known [17, 25].

**Proposition 8**

1. *If a semantics for* $\Sigma$ *is adequate, then* $[\![\phi]\!] \le [\![e]\!][\![\Delta]\!]$ *implies* $\Delta \models e : \phi$.
2. *If a semantics for* $\Sigma$ *is expressive and adequate, then* $[\![\phi]\!] \le [\![e]\!][\![\Delta]\!]$ *iff* $\Delta \models e : \phi$.

**Proof** The £rst part of this proof is a straightforward correctness proof, and follows by induction on $e$.

The proof of the second part begins by showing by induction on $\phi$ that if $\models e : \phi$ then $[\![\phi]\!] \le [\![e]\!]\bot$. This requires expressiveness, for example to prove the case when $\phi = \psi \Rightarrow \chi$ we reason:

$$\begin{array}{lll}
\models \lambda x\,.\,e : \psi \Rightarrow \chi && \\
\quad \Rightarrow \ \models (\lambda x\,.\,e)(\mathsf{term}\,\psi) : \chi && \text{(Expressiveness)} \\
\quad \Rightarrow \ [\![\chi]\!] \le [\![\lambda\,.\,e]\!][\![\mathsf{term}\,\psi]\!] && \text{(Induction)} \\
\quad \Rightarrow \ [\![\chi]\!] \le [\![\lambda\,.\,e]\!][\![\psi]\!] && \text{(Expressiveness)} \\
\quad \Rightarrow \ [\![\psi\Rightarrow\chi]\!] \le [\![\lambda\,.\,e]\!]\bot && \text{(Defn of } \Rightarrow)
\end{array}$$

The result then follows. □

We can combine these propositions to prove full abstraction for CMML.

**Theorem 9 (full abstraction)**

1. *If a semantics for* $\Sigma$ *is adequate, then its extension to* $\mathsf{CMML}\,\Sigma$ *is correct.*

$$\frac{}{* : \mathcal{L} I} \qquad \frac{\phi : \mathcal{L}\sigma \quad \psi : \mathcal{L}\tau}{(\phi,\psi) : \mathcal{L}(\sigma \otimes \tau)} \qquad \frac{}{|a| : \mathcal{L}[A]}[a \in [\![A]\!], a \text{ is compact}]$$

$$\frac{}{\omega : \mathcal{L}(\mathsf{C}\tau)} \qquad \frac{\phi : \mathcal{L}(\mathsf{C}\tau) \quad \psi : \mathcal{L}(\mathsf{C}\tau)}{\phi \wedge \psi : \mathcal{L}(\mathsf{C}\tau)} \qquad \frac{\phi : \mathcal{L}\tau}{[\phi] : \mathcal{L}(\mathsf{C}\tau)}$$

$$\frac{}{\omega : \mathcal{L}(\sigma \to \mathsf{C}\tau)} \qquad \frac{\phi : \mathcal{L}(\sigma \to \mathsf{C}\tau) \quad \psi : \mathcal{L}(\sigma \to \mathsf{C}\tau)}{\phi \wedge \psi : \mathcal{L}(\sigma \to \mathsf{C}\tau)} \qquad \frac{\phi : \mathcal{L}\sigma \quad \psi : \mathcal{L}(\mathsf{C}\tau)}{\phi \Rightarrow \psi : \mathcal{L}(\sigma \to \mathsf{C}\tau)}$$

$$\frac{c \in [\![\mathsf{chan}]\!] \quad \phi : \mathcal{L}(\sigma \to \mathsf{C}\tau)}{\langle c?_\sigma \rangle \phi : \mathcal{L}(\mathsf{C}\tau)} \qquad \frac{c \in [\![\mathsf{chan}]\!] \quad \phi : \mathcal{L}(\sigma \otimes \mathsf{C}\tau)}{\langle c!_\sigma \rangle \phi : \mathcal{L}(\mathsf{C}\tau)} \qquad \frac{\phi : \mathcal{L}(\tau \otimes \mathsf{C}\tau)}{\langle \surd \rangle \phi : \mathcal{L}(\mathsf{C}\tau)}$$

Table 5: The type system for the progam logic

$$\frac{}{\models * : *} \qquad \frac{\models e : \phi \quad \models f : \psi}{\models (e,f) : (\phi,\psi)} \qquad \frac{a \leq [\![\vdash e : [A]]\!]}{\models e : |a|} \qquad \frac{}{\models e : \omega} \qquad \frac{\models e : \phi \quad \models e : \psi}{\models e : \phi \wedge \psi} \qquad \frac{e \xrightarrow{\surd f} e' \quad \models (f, e' \parallel [g]) : \phi}{\models e : \langle \surd \rangle \phi}$$

$$\frac{e \xrightarrow{\acute{}} e' \quad \models e' : \phi}{\models e : \phi} \qquad \frac{\forall \models f : \phi . \models ef : \psi}{\models e : \phi \Rightarrow \psi} \qquad \frac{e \xrightarrow{c!_\sigma f} e' \quad \models (f, e') : \phi}{\models e : \langle c!_\sigma \rangle \phi} \qquad \frac{e \xrightarrow{c?_\sigma x} e' \quad \models \lambda x . e' : \phi}{\models e : \langle c?_\sigma \rangle \phi}$$

Table 6: The operational characteriation of the program logic

$$\begin{aligned}
\mathsf{term}_I * &= * \\
\mathsf{term}_{\sigma\otimes\tau}(\phi,\psi) &= (\mathsf{term}_\sigma \phi, term_\tau \psi) \\
\mathsf{term}_{[A]} |a| &= \mathsf{is}_a \\
\mathsf{term}_{\mathsf{C}\tau} \omega &= \delta \\
\mathsf{term}_{\mathsf{C}\tau}(\phi \wedge \psi) &= \mathsf{term}_{\mathsf{C}\tau} \phi \,\square\, \mathsf{term}_{\mathsf{C}\tau} \psi \\
\mathsf{term}_{\mathsf{C}\tau}[\phi] &= [\mathsf{term}_\tau \phi] \\
\mathsf{term}_{\sigma\to\mathsf{C}\tau} \omega &= \lambda x . \delta \\
\mathsf{term}_{\sigma\to\mathsf{C}\tau}(\phi \wedge \psi) &= \lambda x . (\mathsf{term}_{\sigma\to\mathsf{C}\tau} \phi) x \,\square\, (\mathsf{term}_{\sigma\to\mathsf{C}\tau} \psi) x \\
\mathsf{term}_{I\to\mathsf{C}\tau}(* \Rightarrow \chi) &= \lambda x . \mathsf{term}_{\mathsf{C}\tau} \chi \\
\mathsf{term}_{\rho\otimes\sigma\to\mathsf{C}\tau}((\psi,\phi) \Rightarrow \chi) &= \lambda x .\ \mathsf{let}\, y \Leftarrow (\mathsf{term}_{\rho\to\mathsf{C}I}(\psi \Rightarrow [*]))(x.\mathrm{L}) \\
&\qquad\ \mathsf{in}(\mathsf{term}_{\sigma\to\mathsf{C}\tau}(\phi \Rightarrow \chi))(x.\mathrm{R}) \\
\mathsf{term}_{[A]\to\mathsf{C}\tau}(|a| \Rightarrow \chi) &= \lambda x . \mathsf{let}\, y \Leftarrow (\mathsf{test}_a x) \,\mathsf{in}\, \mathsf{term}_{\mathsf{C}\tau} \chi \\
\mathsf{term}_{\sigma\to\mathsf{C}\tau}(\omega \Rightarrow \chi) &= \lambda x . \mathsf{term}_{\mathsf{C}\tau} \chi \\
\mathsf{term}_{\sigma\to\mathsf{C}\tau}(\phi \wedge \psi \Rightarrow \chi) &= \lambda x .\ \mathsf{let}\, y \Leftarrow \mathsf{term}_{\sigma\to\mathsf{C}I}(\phi \Rightarrow [*]) x \\
&\qquad\ \mathsf{in}\, \mathsf{term}_{\sigma\to\mathsf{C}\tau}(\psi \Rightarrow \chi) x \\
\mathsf{term}_{\mathsf{C}\sigma\to\mathsf{C}\tau}([\phi] \Rightarrow \chi) &= \lambda x . \mathsf{let}\, y \Leftarrow x \,\mathsf{in}\, \mathsf{term}_{\sigma\to\mathsf{C}\tau} y \\
\mathsf{term}_{(\rho\to\mathsf{C}\sigma)\to\mathsf{C}\tau}((\phi \Rightarrow \psi) \Rightarrow \chi) &= \lambda x . (\mathsf{term}_{\mathsf{C}\sigma\to\mathsf{C}\tau}(\psi \Rightarrow \chi))(x(\mathsf{term}_\rho \phi))
\end{aligned}$$

Table 7: Expressiveness result for CMML

2. *If a semantics for $\Sigma$ is expressive and adequate then its extension to* $\mathsf{CMML}\,\Sigma$ *is fully abstract.*

**Proof** Follows from the results that:

- $[\![\tau]\!]$ is algebraic,
- the compact elements of $[\![\tau]\!]$ are characterized precisely as the denotations $[\![\phi]\!]$ of formulae of type $\phi : \mathcal{L}\tau$, and
- the operational and denotation characterizations of when a term satis£es a formula are equivalent $\square$

## 8 Conclusions

This paper has shown that it is possible to combine some of the categorical structure used in giving denotational semantics of functional programming languages with the operational view of programs used to model process algebras.

In the full paper, the monadic strucuture is shown to be exactly the structure required to give denotational models for a programming language with monadic types. There is also a translation of a subset of CML into CMML, based on Moggi's translation of the call-by-value $\lambda$-calculus into MML.

There are a number of outstanding issues for this language:

- Is there a fully abstract semantics for CMML with unique name generation? (This is the most important feature of CML or CHOCS missing from CMML.)
- Is there a fully abstract semantics for must-testing [9] based on acceptance trees [9] or failures sets [11]?

## Acknowledgements

This work is funded by SERC project GR/H 16537, and is carried out in the context of Esprit BRA 7166 Concur 2. Many thanks to Bill Ferreira, Andy Gordon, Matthew Hennessy, Andy Pitts and Edmund Robinson for discussions on this material.

## References

[1] ISO 8807. *LOTOS—A formal description technique based on the temporal ordering of observational behaviour*, 1989.

[2] Samson Abramsky. The lazy lambda calculus. In David Turner, editor, *Declarative Programming*. Addison-Wesley, 1989.

[3] Samson Abramsky. Domain theory in logical form. *Ann. Pure Appl. Logic*, 51:1–77, 1991.

[4] Roberto M. Amadio. Translating core facile. Technical Report ECRC–1994–3, ECRC, 1994.

[5] Dominique Bolignano and Mourad Debabi. A semantic theory for concurrent ML. In *Proc. TACS '94*, 1994.

[6] Andrew Gordon. *Functional Programming and Input/Output*. Ph.D thesis, Cambridge University, 1992.

[7] Andrew Gordon et al. A proposal for monadic i/o in Haskell 1.3. WWW document, Haskell 1.3 Committee, http://www.cl.cam.ac.uk/users/adg/io.html, 1994.

[8] Andrew D. Gordon. Bisimilarity as a theory of functional programming. Submitted to MFPS 95, 1994.

[9] M. Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.

[10] Matthew Hennessy. A denotational model for higher-order processes. Technical Report 6/92, University of Sussex, 1992.

[11] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[12] Douglas Howe. Proving congruence of simulation orderings in functional languages. Unpublished manuscript, 1989.

[13] Douglas J. Howe. Equality in lazy computation systems. In *Proc. LICS 89*, pages 198–203, 1989.

[14] P. Hudak, S. L. Peyton Jones, P. Wadler, et al. A report on the functional language Haskell. *SIGPLAN Notices*, 1992.

[15] Alan Jeffrey. A fully abstract semantics for a higher-order functional concurrent language. A draft copy is available in ftp://ftp.cogs.susx.ac.uk/pub/users/alanje/cmml/draft.ps, 1994.

[16] S. Mac Lane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics. Springer-Verlag, 1971.

[17] Robin Milner. Fully abstract semantics of typed $\lambda$-calculi. *Theoret. Comput. Sci.*, 4:1–22, 1977.

[18] Robin Milner. Calculi for synchrony and asynchrony. *Theoret. Comput. Sci.*, pages 267–310, 1983.

[19] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[20] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes. Technical reports ECS-LFCS-89-86 and -87, LFCS, University of Edinburgh, 1989.

[21] Robin Milner, Mads Tofte, and Robert Harper. *The De£nition of Standard ML*. MIT Press, 1990.

[22] Eugenio Moggi. Notions of computation and mondad. *Inform. and Computing*, 93:55–92, 1991.

[23] C.-H. Luke Ong. *The Lazy Lambda Calculus: An Investigation into the Foundations of Functional Programming*. PhD thesis, Imperial College, London University, 1988.

[24] A. M. Pitts and I. D. B. Stark. On the observable properties of higher order functions that dynamically create local names (preliminary report). In *Workshop on State in Programming Languages, Copenhagen, 1993*, pages 31–45. ACM SIGPLAN, 1993. Yale Univ. Dept. Computer Science Technical Report YALEU/DCS/RR-968.

[25] Gordon Plotkin. LCF considered as a programming language. *Theoret. Comput. Sci.*, 5:223–256, 1977.

[26] J. H. Reppy. A higher-order concurrent langauge. In *Proc. SIGPLAN 91*, pages 294–305, 1991.

[27] J. H. Reppy. *Higher-Order Concurrency*. Ph.D thesis, Cornell University, 1992.

[28] B. Thomsen, L. Leth, S. Prasad, T. M. Kuo, A. Kramer, F. Knabe, and A. Giacalone. Facile antigua release programming guide. Technical Report 93–20, ECRC, 1993.

[29] Bent Thomsen. A calculus of higher order communicating systems. In *Proc. POPL '89*, pages 143–154, 1989.

[30] Bent Thomsen. *Calculi for Higher-Order Communicating Systems*. Ph.D thesis, Imperial College, 1990.

[31] Philip Wadler. Comprehending monads. In *Proc. 1990 ACM Conf. Lisp and Functional Programming*, pages 61–78, 1990.