UNIVERSITY OF SUSSEX

COMPUTER SCIENCE

# Semantics for core Concurrent ML using computation types

Alan Jeffrey

Report 3/96

May 1996

Computer Science
School of Cognitive and Computing Sciences
University of Sussex
Brighton BN1 9QH

ISSN 1350–3170

## Semantics for core Concurrent ML using computation types

ALAN JEFFREY

ABSTRACT. This paper presents two typed higher-order concurrent functional programming languages, based on Reppy's Concurrent ML. The first is a simplified, monomorphic variant of CML, which allows reduction of terms of any type. The second uses an explicit type constructor for computation, in the style of Moggi's monadic metalanguage. Each of these languages is given an operational semantics, which can be used as the basis of bisimulation equivalence. We show how Moggi's translation of the call-by-value lambda-calculus into the mondadic metalanguage can be extended to these concurrent languages, and that this translation is correct up to weak bisimulation.

### 1 Introduction

Reppy's (1991, 1992) Concurrent ML is an extension of New Jersey ML with features for spawning threads, which can communicate by one-to-one synchronous handshake in the style of Milner's (1989) CCS.

There are (at least) two approaches to giving the operational semantics to CML. The 'functional language definition' tradition (Milner, Tofte, and Harper 1990, for example) is to define unlabelled reductions between entire programs, and to use this semantics to prove properties such as type-safety. Reppy uses this approach to give a reduction semantics to CML based on *evaluation contexts* $E[\_]$, for example giving the semantics of if-expressions as:

$$\frac{}{E[\mathtt{if}\,\mathtt{true}\,\mathtt{then}\,e\,\mathtt{else}\,f] \longrightarrow E[e]} \qquad \frac{}{E[\mathtt{if}\,\mathtt{false}\,\mathtt{then}\,e\,\mathtt{else}\,f] \longrightarrow E[f]}$$

The 'concurrency semantics' tradition (Milner 1989, for example) is to define labelled reductions between program fragments, and to use this semantics as the basis of equivalences (such as bisimulation) between program fragments. Ferreira, Hennessy and Jeffrey (1995) use this approach to give a labelled transition system semantics to CML including *silent transitions* $\xrightarrow{\tau}$ and *value transitions* $\xrightarrow{\surd v}$, for example giving the semantics of if-expressions as:

$$\frac{e \xrightarrow{\surd \mathtt{true}} e'}{\mathtt{if}\,e\,\mathtt{then}\,f\,\mathtt{else}\,g \xrightarrow{\tau} e' \,\|\, f} \qquad \frac{e \xrightarrow{\surd \mathtt{false}} e'}{\mathtt{if}\,e\,\mathtt{then}\,f\,\mathtt{else}\,g \xrightarrow{\tau} e' \,\|\, g}$$

$$\frac{e \xrightarrow{\alpha} e'}{\mathtt{if}\,e\,\mathtt{then}\,f\,\mathtt{else}\,g \xrightarrow{\alpha} \mathtt{if}\,e'\,\mathtt{then}\,f\,\mathtt{else}\,g}$$

Presented at *Higher Order Operational Techniques in Semantics* workshop, Newton Institute, Cambridge, October 1995.

The resulting labelled transition system can be used as the basis of an equational theory of CML expressions, using *bisimulation* as equivalence.

Unfortunately, there are some problems with this semantics:

- It is complex, due to having to allow expressions in any evaluation context to reduce (for example requiring three rules for if-expressions rather than Reppy's two axiom schemas).
- It produces very long reductions, due to large numbers of 'book-keeping' steps (for example the long reduction in Table 9).
- The resulting equational theory does not have pleasant mathematical properties (for example neither β- nor η-conversion hold for the language).

In this paper we present a variant of CML using *computation types*. These provide an explicit type constructor `_comp` for computation, which means that the type system can distinguish between expressions which can perform computation (those of type $A\,\mathsf{comp}$) and those which are guaranteed to be in normal form (anything else). Differentiating by type between expressions which can and cannot perform reductions makes the operational semantics much simpler, for example the much shorter reduction in Table 16 and the simpler operational rules for if-expressions:

$$\frac{}{\mathsf{if\ true\ then}\ f\ \mathsf{else}\ g \xrightarrow{\tau} f} \qquad \frac{}{\mathsf{if\ false\ then}\ f\ \mathsf{else}\ g \xrightarrow{\tau} g}$$

Computation types were originally proposed by Moggi (1991) in a denotational setting to provide models of non-trivial computation (such as CML communication) without losing pleasant mathematical properties (such as β- and η-reduction). Moggi provided a translation from the call-by-value λ-calculus into the language with computation types, which we can adapt for CML and prove to be correct up to weak bisimulation.

We can also use equational reasoning to transform inefficient programs (such as the translation of the long reduction in Table 9) into efficient ones (such as the short reduction in Table 16). We conjecture that such optimizations may make languages with explicit computation types simpler to optimize.

IN SECTION 2 we present a cut-down version of the operational semantics for CML presented in (Ferreira, Hennessy, and Jeffrey 1995), including a suitable definition of bisimulation for CML programs.

IN SECTION 3 we present the variant of CML with explicit computation types, and show that the resulting equational theory of bisimulation has better mathematical properties than that of CML. This is a variant of the language presented in (Jeffrey 1995a).



IN SECTION 4 we provide a translation from the first language into the second, and show that it is correct up to bisimulation.

## 2 Concurrent ML

In this section, we introduce a subset of Concurrent ML (CML), and provide a labelled transition system semantics for it. This provides weak bisimulation as an equivalence on programs.

This section is based on joint work with Ferreira and Hennessy, and is discussed in more detail in (Ferreira, Hennessy, and Jeffrey 1995).

### *2.1 Syntax*

Concurrent ML (CML) is an extension to New Jersey ML which allows for the implementation of concurrent programs. Communication takes place along channels, and is a one-to-one handshake similar to Milner's (1989) CCS. For example, the process which transmits value $v$ of type $A$ along channel `a` and then returns the canonical value `()` of type `unit` is:

$$\mathtt{send}_A\,(\mathtt{a}, v)$$

In this paper, we are using a simplified notion of channel, where channels are untyped, and so $\mathtt{send}_A$ has type:

$$\mathtt{send}_A : (\mathtt{chan} * A) \mathtt{->} \mathtt{unit}$$

The process which accepts value $v$ of type $A$ along channel `a` and returns $v$ is:

$$\mathtt{accept}_A\ \mathtt{a}$$

This has type:

$$\mathtt{accept}_A : \mathtt{chan} \mathtt{->} A$$

The fragment of CML we are considering is monomorphic, which is why `send` and `accept` have to be type-indexed. We shall often elide these indices.

Evaluation proceeds as in ML, with left-to-right call-by-value evaluation, so a process which accepts values $v$ then $w$ along channel `a` and returns the pair $(v, w)$ is:

$$(\mathtt{accept\ a}, \mathtt{accept\ a})$$

We can define the sequential composition of $e$ and $f$ to be a term which evaluates $e$, discards the result, then evaluates $f$ to be (for fresh $x$):

$$e ; f = \mathtt{let}\ x = e\ \mathtt{in}\ f$$

A thunked process can be forked off for concurrent evaluation using `spawn`, for example the concurrent passing of $v$ along `a` can be given:

$$\mathtt{spawn}\,(\mathtt{fn}\ x \Rightarrow \mathtt{send}\,(\mathtt{a}, v)) ; \mathtt{accept\ a}$$

This spawns $\mathtt{send}\,(\mathtt{a},v)$ off for concurrent execution, then evaluates $\mathtt{accept}\ \mathtt{a}$. These two processes can then communicate. In this paper, we are ignoring CML's *threads* so `spawn` has type:

$$\mathtt{spawn} : (\mathtt{unit} \to A) \to \mathtt{unit}$$

CML does *not* provide a general 'external choice' operator such as CCS $+$. Instead, guarded choice is provided, and the type mechanism is used to ensure that choice is only ever used on guarded computation. The type $A\ \mathtt{event}$ is used as the type of guarded processes of type $A$, and CML allows for the creation of guarded input and output:

$$\mathtt{transmit}_A : (\mathtt{chan} * A) \to \mathtt{unit}\ \mathtt{event} \qquad \mathtt{receive}_A : \mathtt{chan} \to A\ \mathtt{event}$$

and for guarded sequential computation:

$$\mathtt{wrap} : (A\ \mathtt{event} * (A \to B)) \to B\ \mathtt{event}$$

For example the guarded process which inputs a value on `a` and outputs it on `b` is given:

$$\mathtt{wrap}\,(\mathtt{receive}\ \mathtt{a}, \mathtt{fn}\ x \Rightarrow \mathtt{send}\,(\mathtt{b}, x)) : \mathtt{unit}\ \mathtt{event}$$

CML provides choice between guarded processes using `choose`. In CML this is defined on lists, but for simplicity we shall give it only for pairs:

$$\mathtt{choose} : (A\ \mathtt{event} * A\ \mathtt{event}) \to A\ \mathtt{event}$$

For example the guarded process which chooses between receiving a signal on `a` or `b` is:

$$\mathtt{choose}\,(\mathtt{receive}_A\ \mathtt{a}, \mathtt{receive}_A\ \mathtt{b}) : A\ \mathtt{event}$$

Guarded processes can be treated as any other process, using the function `sync`:

$$\mathtt{sync} : A\ \mathtt{event} \to A$$

For example, we can execute the above guarded process by saying:

$$\mathtt{sync}\,(\mathtt{choose}\,(\mathtt{receive}_A\ \mathtt{a}, \mathtt{receive}_A\ \mathtt{b})) : A$$

In fact, `accept` and `send` are not primitives in CML, and are defined:

$$\mathtt{accept}_A \stackrel{\mathsf{def}}{=} \mathtt{fn}\ x \Rightarrow \mathtt{sync}\,(\mathtt{receive}_A\ x)$$
$$\mathtt{send}_A \stackrel{\mathsf{def}}{=} \mathtt{fn}\ x \Rightarrow \mathtt{sync}\,(\mathtt{transmit}_A\ x)$$

This paper cannot provide a full introduction to CML, and the interested reader is referred to Reppy's papers (Reppy 1991; Reppy 1992) for further explanation.

The fragment of CML we will consider here is missing much of CML's functionality, notably polymorphism, guards and thread identifiers. It is similar to the fragment of CML considered in (Ferreira, Hennessy, and Jeffrey 1995) except



$$\frac{\Gamma \vdash e : A}{\Gamma \vdash ce : B}[c : A \to B] \qquad \frac{\Gamma \vdash e : \mathtt{bool} \quad \Gamma \vdash f : A \quad \Gamma \vdash g : A}{\Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ f\ \mathtt{else}\ g : A}$$

$$\frac{\Gamma \vdash e : A \quad \Gamma \vdash f : B}{\Gamma \vdash (e, f) : A * B} \qquad \frac{\Gamma \vdash e : A \quad \Gamma, x : A \vdash f : B}{\Gamma \vdash \mathtt{let}\ x = e\ \mathtt{in}\ f : B}$$

$$\frac{\Gamma \vdash e : A \to B \quad \Gamma \vdash f : A}{\Gamma \vdash ef : B} \qquad \frac{}{\Gamma, x : A \vdash x : A} \qquad \frac{\Gamma \vdash x : A}{\Gamma, y : B \vdash x : A}[x \neq y]$$

$$\frac{}{\Gamma \vdash \mathtt{true} : \mathtt{bool}} \qquad \frac{}{\Gamma \vdash \mathtt{false} : \mathtt{bool}} \qquad \frac{}{\Gamma \vdash n : \mathtt{int}} \qquad \frac{}{\Gamma \vdash k : \mathtt{chan}}$$

$$\frac{}{\Gamma \vdash () : \mathtt{unit}} \qquad \frac{\Gamma, x : A \to B, y : A \vdash e : B}{\Gamma \vdash \mathtt{rec}\ x = \mathtt{fn}\ y \Rightarrow e : A \to B}$$

TABLE 1. Types for $\mu$CML expressions

that for simplicity we do not consider the `always` command. We will call this subset 'core $\tau$-free CML', or $\mu$CML for short.

For simplicity, we will only use `unit`, `bool`, `int` and `chan` as base types, although other types such as lists could easily be added.

The *integer values* are given by the grammar:

$$n ::= \cdots \mid \mathtt{-1} \mid \mathtt{0} \mid \mathtt{1} \mid \cdots$$

The *channel values* are given by the grammar:

$$k ::= \mathtt{a} \mid \mathtt{b} \mid \cdots$$

The *values* are given by the grammar:

$$v ::= \mathtt{true} \mid \mathtt{false} \mid n \mid k \mid () \mid \mathtt{rec}\ x = \mathtt{fn}\ x \Rightarrow e \mid x$$

The *expressions* are given by the grammar:

$$e ::= v \mid ce \mid \mathtt{if}\ e\ \mathtt{then}\ e\ \mathtt{else}\ e \mid (e, e) \mid \mathtt{let}\ x = e\ \mathtt{in}\ e \mid ee$$

Finally, the *basic functions* are given by the grammar:

$$c ::= \mathtt{fst} \mid \mathtt{snd} \mid \mathtt{add} \mid \mathtt{mul} \mid \mathtt{leq} \mid \mathtt{transmit}_A \mid \mathtt{receive}_A$$
$$\mid \mathtt{choose} \mid \mathtt{spawn} \mid \mathtt{sync} \mid \mathtt{wrap} \mid \mathtt{never}$$

$\mu$CML is a typed language, with a type system given by the grammar:

$$A ::= \mathtt{unit} \mid \mathtt{bool} \mid \mathtt{int} \mid \mathtt{chan} \mid A * A \mid A \to A \mid A\ \mathtt{event}$$

The type judgements for expressions are given as judgements $\Gamma \vdash e : A$, where $\Gamma$ ranges over contexts of the form $x_1 : A_1, \ldots, x_n : A_n$. The type system is in Tables 1 and 2.

We can define syntactic sugar for $\mu$CML definitions, writing $\mathtt{fn}\ x \Rightarrow e$ for $\mathtt{rec}\ y = \mathtt{fn}\ x \Rightarrow e$ when $y$ is not free in $e$, using pattern-matching on pairs as

$$
\begin{aligned}
\texttt{fst} &: A * B \rightarrow A \\
\texttt{snd} &: A * B \rightarrow B \\
\texttt{add} &: \texttt{int} * \texttt{int} \rightarrow \texttt{int} \\
\texttt{mul} &: \texttt{int} * \texttt{int} \rightarrow \texttt{int} \\
\texttt{leq} &: \texttt{int} * \texttt{int} \rightarrow \texttt{bool} \\
\texttt{transmit}_A &: \texttt{chan} * A \rightarrow \texttt{unit event} \\
\texttt{receive}_A &: \texttt{chan} \rightarrow A\ \texttt{event} \\
\texttt{choose} &: A\ \texttt{event} * A\ \texttt{event} \rightarrow A\ \texttt{event} \\
\texttt{spawn} &: \texttt{unit -> unit} \rightarrow \texttt{unit} \\
\texttt{sync} &: A\ \texttt{event} \rightarrow A \\
\texttt{wrap} &: A\ \texttt{event} * (A\ \texttt{->}\ B) \rightarrow B\ \texttt{event} \\
\texttt{never} &: \texttt{unit} \rightarrow A\ \texttt{event}
\end{aligned}
$$

TABLE 2. Types for $\mu$CML basic functions

shorthand for projections, and using $\stackrel{\text{def}}{=}$ as shorthand for recursive function declaration. For example, a one-place buffer can be defined:

$$
\begin{aligned}
\texttt{cell}_A &: \texttt{chan} * \texttt{chan ->} B \\
\texttt{cell}_A\,(x\,,y) &\stackrel{\text{def}}{=} \texttt{cell}_A\,(\texttt{snd}\,(\texttt{send}_A\,(y\,,\texttt{accept}_A\,x)\,,(x\,,y)))
\end{aligned}
$$

### 2.2 *Operational semantics*

The semantics we will use here is based on the 'semantics of concurrency' tradition: we extend the programming language with enough syntactic constructs that it is possible to give a transition system semantics between program fragments. A comparison of this semantics with Reppy's (1992) reduction semantics is given in (Ferreira, Hennessy, and Jeffrey 1995).

The semantics we provide has four transitions: reduction ($\tau$), returning a value ($\checkmark v$), input on a channel ($k?x$), and output on a channel ($k!v$).

A transition $e \xrightarrow{\tau} e'$ represents a single-step reduction, for example[1]:

$$\texttt{if true then 0 else 1} \xrightarrow{\tau} \texttt{0}$$

We will often write $e \Longrightarrow e'$ for $e \xrightarrow{\tau} \cdots \xrightarrow{\tau} e'$, for example:

$$\texttt{if true then add(1,-1) else 1} \Longrightarrow \texttt{0}$$

[1] In this example, and in others, we have 'garbage collected' some empty processes by treating $\|$ as an associative operation with left unit $\delta$. These equivalences are correct up to strong bisimulation.



A transition $e \xrightarrow{\checkmark v} e'$ represents a process returning a value $v$, for example:

$$\texttt{0} \xrightarrow{\checkmark \texttt{0}} \delta$$

We will often write $e \stackrel{l}{\Longrightarrow} e'$ for $e \Longrightarrow \xrightarrow{l} e'$, for example:

$$\texttt{if true then add(1,-1) else 1} \stackrel{\checkmark \texttt{0}}{\Longrightarrow} \delta$$

In this case the computation is sequential, so the remaining computation after returning the value '0' is the empty computation '$\delta$'. CML allows processes to spawn threads which can continue after their parent has terminated, so there are cases when the remaining computation is non-trivial, such as:

$$\texttt{spawn (fn () => send (a,0))} \stackrel{\checkmark \texttt{()}}{\Longrightarrow} \texttt{send (a,0)} \,\|\, \delta$$

Here '$\|$' represents the parallel composition of two processes, with the rightmost process being the main thread of computation, for example:

$$\texttt{spawn (fn () => send (a,0));accept a} \Longrightarrow \texttt{send (a,0)} \,\|\, \texttt{accept a}$$

A transition $e \xrightarrow{k?x} e'$ represents an input on channel $k$, where $e'$ has a free variable $x$, for example:

$$\texttt{accept a} \stackrel{\texttt{a?x}}{\Longrightarrow} x$$

Similarly, a transition $e \xrightarrow{k!v} e'$ represents an output of value $v$ on channel $k$, for example:

$$\texttt{send (a,0)} \stackrel{\texttt{a!0}}{\Longrightarrow} \texttt{()}$$

Input and output transitions can be synchronized to produce reductions, for example:

$$\texttt{send (a,0)} \,\|\, \texttt{accept a} \Longrightarrow \texttt{()} \,\|\, \texttt{0}$$

In $\mu$CML there are no normal forms for pairs—such a normal form is needed for the operational semantics, so we will extend the language of values with pairs $\langle v\,,w\rangle$. This allows pairs of values to be communicated, for example since:

$$\texttt{(1,-1)} \stackrel{\checkmark \langle \texttt{1,-1}\rangle}{\Longrightarrow} \delta$$

we have:

$$\texttt{send (b,(1,-1))} \stackrel{\texttt{b!}\langle \texttt{1,-1}\rangle}{\Longrightarrow} \texttt{()}$$

and so we have the communication:

$$\texttt{send (b,(1,-1))} \,\|\, \texttt{add (accept b)} \Longrightarrow \texttt{()} \,\|\, \texttt{add}\langle \texttt{-1,1}\rangle$$

So far we have only considered first-order processes, but CML is a higher-order language which can communicate values of any type, for example since:

$$\texttt{send} \stackrel{\checkmark\, \texttt{send}}{\Longrightarrow} \delta$$

we have:

$$\mathtt{send\,(b,send\,)} \xrightarrow{\mathtt{b!send}} \mathtt{()}$$

and so we have the higher-order communication:

$$\mathtt{send\,(b,send\,)} \parallel \mathtt{accept\,b(a,0)} \Longrightarrow \mathtt{()} \parallel \mathtt{send\,(a,0)}$$

CML also allows communications of events, so we need to extend the language in a similar fashion to Reppy (1992) to include values of event type. These values are of the form $[ge]$ where $ge$ is a CCS-style *guarded sum*, for example:

$$\begin{aligned}
\mathtt{transmit\,(a,0)} &\Longrightarrow [\mathtt{a!0}] \\
\mathtt{receive\,a} &\Longrightarrow [\mathtt{a?}] \\
\mathtt{choose\,(transmit\,(a,0),receive\,a)} &\Longrightarrow [\mathtt{a!0} \oplus \mathtt{a?}] \\
\mathtt{wrap\,(receive\,a,fn}\,x\,\mathtt{=>}\,e\,) &\Longrightarrow [\mathtt{a?} \Rightarrow \mathtt{fn}\,x\,\mathtt{=>}\,e]
\end{aligned}$$

This syntax is based on Reppy's, and is slightly different from that normally associated with process calculi, for example:

- we write $\mathtt{a!0} \oplus \mathtt{a?}$ rather than $\mathtt{a!0} + \mathtt{a?}$, and
- we write $\mathtt{a?} \Rightarrow \mathtt{fn}\,x\,\mathtt{=>}\,e$ rather than $\mathtt{a?}x.e$.

By extending the syntax of $\mu$CML expressions to include guarded expressions, we get a particularly simple semantics for `sync` as just removing the outermost level of `[_]`, for example:

$$\begin{aligned}
&\mathtt{send\,(a,0)} \\
&\quad\Longrightarrow \mathtt{sync\,(transmit\,(a,0))} \\
&\quad\Longrightarrow \mathtt{sync\,[a!0]} \\
&\quad\Longrightarrow \mathtt{a!0} \\
&\quad\xrightarrow{\mathtt{a!0}} \mathtt{()}
\end{aligned}$$

In summary, we give the operational semantics for $\mu$CML by first extending it to $\mu\text{CML}^+$ by adding expressions:

$$e ::= \cdots \mid e \parallel e \mid ge$$

adding values:

$$v ::= \cdots \mid \langle v, v \rangle \mid [ge]$$

and adding guarded expressions:

$$ge ::= k?_A \mid k!_A v \mid \delta \mid ge \oplus ge \mid ge \Rightarrow v$$

The typing for $\mu\text{CML}^+$ extends that of $\mu$CML with the rules in Table 3.

The extended language $\mu\text{CML}^+$ has a semantics as a labelled transition system with labels:

$$\mu ::= k!_A v \mid k?_A x \qquad \alpha ::= \tau \mid \mu \qquad l ::= \alpha \mid \checkmark v$$



$$\frac{\Gamma \vdash e : A \quad \Gamma \vdash f : B}{\Gamma \vdash e \parallel f : B} \qquad \frac{}{\Gamma \vdash \delta : A}$$

$$\frac{\Gamma \vdash v : A \quad \Gamma \vdash w : B}{\Gamma \vdash \langle v, w \rangle : A * B} \qquad \frac{\Gamma \vdash e : A}{\Gamma \vdash [e] : A\ \mathtt{event}}$$

$$\frac{\Gamma \vdash v : \mathtt{chan} \quad \Gamma \vdash w : A}{\Gamma \vdash v!_A w : \mathtt{unit}} \qquad \frac{\Gamma \vdash v : \mathtt{chan}}{\Gamma \vdash v?_A : A}$$

$$\frac{\Gamma \vdash ge_1 : A \quad \Gamma \vdash ge_2 : A}{\Gamma \vdash ge_1 \oplus ge_2 : A} \qquad \frac{\Gamma \vdash ge : A \quad \Gamma \vdash v : A \mathtt{->} B}{\Gamma \vdash ge \Rightarrow v : B}$$

TABLE 3. Types for $\mu\text{CML}^+$ expressions

$$\frac{e \xrightarrow{\alpha} e'}{ce \xrightarrow{\alpha} ce'} \qquad \frac{e \xrightarrow{\alpha} e'}{\mathtt{if}\ e\ \mathtt{then}\ f\ \mathtt{else}\ g \xrightarrow{\alpha} \mathtt{if}\ e'\ \mathtt{then}\ f\ \mathtt{else}\ g}$$

$$\frac{e \xrightarrow{\alpha} e'}{(e,f) \xrightarrow{\alpha} (e',f)} \qquad \frac{e \xrightarrow{\alpha} e'}{\mathtt{let}\ x = e\ \mathtt{in}\ f \xrightarrow{\alpha} \mathtt{let}\ x = e'\ \mathtt{in}\ f}$$

$$\frac{e \xrightarrow{\alpha} e'}{ef \xrightarrow{\alpha} e'f} \qquad \frac{e \xrightarrow{\alpha} e'}{e \parallel f \xrightarrow{\alpha} e' \parallel f} \qquad \frac{f \xrightarrow{l} f'}{e \parallel f \xrightarrow{l} e \parallel f'}$$

TABLE 4. CML operational semantics: static rules

The operational semantics is given in Tables 4–8.

This operational semantics is very fine-grained, and is designed to mimic the execution of a CML program very closely. As a result, derivations of fairly simple computations can be surprisingly long. For example, one reduction of $\mathtt{cell}_A\langle i, o\rangle$ is given in Table 9.

### 2.3 Bisimulation

As we mentioned above, one reason for choosing a labelled transition system semantics over a reduction semantics is that we can define *bisimulation* as an equivalence on programs. This is discussed at length in (Ferreira, Hennessy, and Jeffrey 1995), and is summarized here. We will use notation adapted from Gordon's (1995) presentation of Howe's (1989) proof technique.

Let an *open type-indexed* relation $\mathcal{R}$ be a family of relations $\mathcal{R}_{\Gamma,A}$ such that if $e\ \mathcal{R}_{\Gamma,A}\ f$ then $\Gamma \vdash e : A$ and $\Gamma \vdash f : A$. We will often elide the subscripts from relations, for example writing $e\ \mathcal{R}\ f$ for $e\ \mathcal{R}_{\Gamma,A}\ f$ when context makes the type

$$\frac{ge_1 \xrightarrow{\alpha} e}{ge_1 \oplus ge_2 \xrightarrow{\alpha} e} \qquad \frac{ge_2 \xrightarrow{\alpha} e}{ge_1 \oplus ge_2 \xrightarrow{\alpha} e} \qquad \frac{ge \xrightarrow{\alpha} e}{ge \Rightarrow v \xrightarrow{\alpha} ve}$$

TABLE 5. CML operational semantics: dynamic rules

$$\frac{e \xrightarrow{\surd v} e'}{ef \xrightarrow{\tau} e' \parallel \mathtt{let}\ y = f\ \mathtt{in}\ g[v/x]}[v = \mathtt{rec}\ x = \mathtt{fn}\ y \Rightarrow g]$$

$$\frac{e \xrightarrow{\surd v} e'}{ce \xrightarrow{\tau} e' \parallel \delta(c, v)}$$

$$\frac{e \xrightarrow{\surd \mathtt{true}} e'}{\mathtt{if}\ e\ \mathtt{then}\ f\ \mathtt{else}\ g \xrightarrow{\tau} e' \parallel f} \qquad \frac{e \xrightarrow{\surd \mathtt{false}} e'}{\mathtt{if}\ e\ \mathtt{then}\ f\ \mathtt{else}\ g \xrightarrow{\tau} e' \parallel g}$$

$$\frac{e \xrightarrow{\surd v} e'}{(e, f) \xrightarrow{\tau} e' \parallel \mathtt{let}\ x = f\ \mathtt{in}\ \langle v, x\rangle} \qquad \frac{e \xrightarrow{\surd v} e'}{\mathtt{let}\ x = e\ \mathtt{in}\ f \xrightarrow{\tau} e' \parallel f[v/x]}$$

$$\frac{e \xrightarrow{k!_A v} e' \quad f \xrightarrow{k?_A x} f'}{e \parallel f \xrightarrow{\tau} e' \parallel f'[v/x]} \qquad \frac{e \xrightarrow{k?_A x} e' \quad f \xrightarrow{k!_A v} f'}{e \parallel f \xrightarrow{\tau} e'[v/x] \parallel f'}$$

TABLE 6. CML operational semantics: silent reductions

$$\frac{}{v \xrightarrow{\surd v} \delta} \qquad \frac{}{k!_A v \xrightarrow{k!_A v} ()} \qquad \frac{}{k?_A \xrightarrow{k?_A x} x}$$

TABLE 7. CML operational semantics: axioms

$$\begin{array}{rlrl}
\delta(\mathtt{fst}, \langle v, w\rangle) = & v & \delta(\mathtt{transmit}_A, \langle k, v\rangle) = & [k!_A v] \\
\delta(\mathtt{snd}, \langle v, w\rangle) = & w & \delta(\mathtt{receive}_A, k) = & [k?_A] \\
\delta(\mathtt{add}, \langle m, n\rangle) = & m + n & \delta(\mathtt{choose}, \langle [ge_1], [ge_2]\rangle) = & [ge_1 \oplus ge_2] \\
\delta(\mathtt{mul}, \langle m, n\rangle) = & m \times n & \delta(\mathtt{wrap}, \langle [ge], v\rangle) = & [ge \Rightarrow v] \\
\delta(\mathtt{leq}, \langle m, n\rangle) = & m \leq n & \delta(\mathtt{spawn}, v) = & v\,() \parallel () \\
\delta(\mathtt{sync}, [ge]) = & ge & \delta(\mathtt{never}, ()) = & [\delta]
\end{array}$$

TABLE 8. CML operational semantics: basic functions



$$\begin{array}{l}
\mathtt{cell}_A \langle i, o\rangle \\
\xrightarrow{\tau} \mathtt{let}\ x = \langle i, o\rangle\ \mathtt{in}\ \mathtt{cell}_A\ (\mathtt{snd}\ (\mathtt{send}_A\ (\mathtt{snd}\ x, \mathtt{accept}_A\ (\mathtt{fst}\ x)), x)) \\
\xrightarrow{\tau} \mathtt{cell}_A\ (\mathtt{snd}\ (\mathtt{send}_A\ (\mathtt{snd}\langle i, o\rangle, \mathtt{accept}_A\ (\mathtt{fst}\langle i, o\rangle)), \langle i, o\rangle)) \\
\xrightarrow{\tau} \mathtt{let}\ x = \mathtt{snd}\ (\mathtt{send}_A\ (\mathtt{snd}\langle i, o\rangle, \mathtt{accept}_A\ (\mathtt{fst}\langle i, o\rangle)), \langle i, o\rangle) \\
\quad \mathtt{in}\ \mathtt{cell}_A\ x \\
\xrightarrow{\tau} \mathtt{let}\ x = \mathtt{snd}\ (\mathtt{let}\ y = (\mathtt{snd}\langle i, o\rangle, \mathtt{accept}_A\ (\mathtt{fst}\langle i, o\rangle)) \\
\qquad\qquad \mathtt{in}\ \mathtt{sync}\ (\mathtt{transmit}_A\ y) \\
\qquad\quad , \langle i, o\rangle) \\
\quad \mathtt{in}\ \mathtt{cell}_A\ x \\
\xrightarrow{\tau} \mathtt{let}\ x = \mathtt{snd}\ (\mathtt{let}\ y = (o, \mathtt{accept}_A\ (\mathtt{fst}\langle i, o\rangle)) \\
\qquad\qquad \mathtt{in}\ \mathtt{sync}\ (\mathtt{transmit}_A\ y) \\
\qquad\quad , \langle i, o\rangle) \\
\quad \mathtt{in}\ \mathtt{cell}_A\ x \\
\xrightarrow{\tau} \mathtt{let}\ x = \mathtt{snd}\ (\mathtt{let}\ y = \mathtt{let}\ z = \mathtt{accept}_A\ (\mathtt{fst}\langle i, o\rangle)\ \mathtt{in}\ \langle o, z\rangle \\
\qquad\qquad \mathtt{in}\ \mathtt{sync}\ (\mathtt{transmit}_A\ y) \\
\qquad\quad , \langle i, o\rangle) \\
\quad \mathtt{in}\ \mathtt{cell}_A\ x \\
\xrightarrow{\tau} \mathtt{let}\ x = \mathtt{snd}\ (\mathtt{let}\ y = \mathtt{let}\ z = \mathtt{let}\ x' = \mathtt{fst}\langle i, o\rangle\ \mathtt{in}\ \mathtt{sync}\ (\mathtt{receive}_A\ x')\ \mathtt{in}\ \langle o, z\rangle \\
\qquad\qquad \mathtt{in}\ \mathtt{sync}\ (\mathtt{transmit}_A\ y) \\
\qquad\quad , \langle i, o\rangle) \\
\quad \mathtt{in}\ \mathtt{cell}_A\ x \\
\xrightarrow{\tau} \mathtt{let}\ x = \mathtt{snd}\ (\mathtt{let}\ y = \mathtt{let}\ z = \mathtt{let}\ x' = i\ \mathtt{in}\ \mathtt{sync}\ (\mathtt{receive}_A\ x')\ \mathtt{in}\ \langle o, z\rangle \\
\qquad\qquad \mathtt{in}\ \mathtt{sync}\ (\mathtt{transmit}_A\ y) \\
\qquad\quad , \langle i, o\rangle) \\
\quad \mathtt{in}\ \mathtt{cell}_A\ x \\
\xrightarrow{\tau} \mathtt{let}\ x = \mathtt{snd}\ (\mathtt{let}\ y = \mathtt{let}\ z = \mathtt{sync}\ (\mathtt{receive}_A\ i)\ \mathtt{in}\ \langle o, z\rangle \\
\qquad\qquad \mathtt{in}\ \mathtt{sync}\ (\mathtt{transmit}_A\ y) \\
\qquad\quad , \langle i, o\rangle) \\
\quad \mathtt{in}\ \mathtt{cell}_A\ x \\
\xrightarrow{\tau} \mathtt{let}\ x = \mathtt{snd}\ (\mathtt{let}\ y = \mathtt{let}\ z = \mathtt{sync}\ [i?_A]\ \mathtt{in}\ \langle o, z\rangle\ \mathtt{in}\ \mathtt{sync}\ (\mathtt{transmit}_A\ y), \langle i, o\rangle) \\
\quad \mathtt{in}\ \mathtt{cell}_A\ x \\
\xrightarrow{\tau} \mathtt{let}\ x = \mathtt{snd}\ (\mathtt{let}\ y = \mathtt{let}\ z = i?_A\ \mathtt{in}\ \langle o, z\rangle\ \mathtt{in}\ \mathtt{sync}\ (\mathtt{transmit}_A\ y), \langle i, o\rangle) \\
\quad \mathtt{in}\ \mathtt{cell}_A\ x \\
\xrightarrow{i?v} \mathtt{let}\ x = \mathtt{snd}\ (\mathtt{let}\ y = \mathtt{let}\ z = v\ \mathtt{in}\ \langle o, z\rangle\ \mathtt{in}\ \mathtt{sync}\ (\mathtt{transmit}_A\ y), \langle i, o\rangle) \\
\quad \mathtt{in}\ \mathtt{cell}_A\ x \\
\xrightarrow{\tau} \mathtt{let}\ x = \mathtt{snd}\ (\mathtt{let}\ y = \langle o, v\rangle\ \mathtt{in}\ \mathtt{sync}\ (\mathtt{transmit}_A\ y), \langle i, o\rangle) \\
\quad \mathtt{in}\ \mathtt{cell}_A\ x \\
\xrightarrow{\tau} \mathtt{let}\ x = \mathtt{snd}\ (\mathtt{sync}\ (\mathtt{transmit}_A \langle o, v\rangle), \langle i, o\rangle)\ \mathtt{in}\ \mathtt{cell}_A\ x \\
\xrightarrow{\tau} \mathtt{let}\ x = \mathtt{snd}\ (\mathtt{sync}\ [o!_A v], \langle i, o\rangle)\ \mathtt{in}\ \mathtt{cell}_A\ x \\
\xrightarrow{\tau} \mathtt{let}\ x = \mathtt{snd}\ (o!_A v, \langle i, o\rangle)\ \mathtt{in}\ \mathtt{cell}_A\ x \\
\xrightarrow{o!v} \mathtt{let}\ x = \mathtt{snd}\ ((), \langle i, o\rangle)\ \mathtt{in}\ \mathtt{cell}_A\ x \\
\xrightarrow{\tau} \mathtt{let}\ x = \mathtt{snd}\ (\mathtt{let}\ y = \langle i, o\rangle\ \mathtt{in}\ \langle (), y\rangle)\ \mathtt{in}\ \mathtt{cell}_A\ x \\
\xrightarrow{\tau} \mathtt{let}\ x = \mathtt{snd}\langle (), \langle i, o\rangle\rangle\ \mathtt{in}\ \mathtt{cell}_A\ x \\
\xrightarrow{\tau} \mathtt{let}\ x = \langle i, o\rangle\ \mathtt{in}\ \mathtt{cell}_A\ x \\
\xrightarrow{\tau} \mathtt{cell}_A \langle i, o\rangle
\end{array}$$

TABLE 9. CML operational semantics: example reduction

obvious.

Let a *closed type-indexed* relation $\mathcal{R}$ be an open type-indexed relation where $\Gamma$ is everywhere the empty context, and can therefore be elided.

For any closed type-indexed relation $\mathcal{R}$, let its *open extension* $\mathcal{R}^\circ$ be defined as:

$$\mathtt{e}\ \mathcal{R}^\circ_{\vec{\mathtt{x}}:\vec{A},B}\ \mathtt{f} \text{ iff } \mathtt{e}[\vec{\mathtt{v}}/\vec{\mathtt{x}}]\ \mathcal{R}_B\ \mathtt{f}[\vec{\mathtt{v}}/\vec{\mathtt{x}}] \text{ for all } \vdash \vec{\mathtt{v}} : \vec{A}.$$

A closed type-indexed relation $\mathcal{R}$ is *structure preserving* iff:

- if $\mathtt{v}\ \mathcal{R}_A\ \mathtt{w}$ and $A$ is a base type then $\mathtt{v} = \mathtt{w}$,
- if $\langle \mathtt{v}_1, \mathtt{v}_2\rangle\ \mathcal{R}_{A_1 * A_2}\ \langle \mathtt{w}_1, \mathtt{w}_2\rangle$ then $\mathtt{v}_i\ \mathcal{R}_{A_i}\ \mathtt{w}_i$,
- if $[ge_1]\ \mathcal{R}_{A\ \mathtt{event}}\ [ge_2]$ then $ge_1\ \mathcal{R}_A\ ge_2$, and
- if $\mathtt{v}\ \mathcal{R}_{A\mathtt{->}B}\ \mathtt{v}'$ then for all $\vdash \mathtt{w} : A$ we have $\mathtt{v}\mathtt{w}\ \mathcal{R}_B\ \mathtt{v}'\mathtt{w}$.

A closed type-indexed relation $\mathcal{R}$ is a *first-order strong simulation* iff it is structure preserving and the following diagram can be completed:

$$\begin{tikzcd} \mathtt{e}_1 \arrow[r, phantom, "\mathcal{R}"] \arrow[d, "l"'] & \mathtt{e}_2 \\ \mathtt{e}_1' & \end{tikzcd} \quad \text{as} \quad \begin{tikzcd} \mathtt{e}_1 \arrow[r, phantom, "\mathcal{R}"] \arrow[d, "l"'] & \mathtt{e}_2 \arrow[d, "l"'] \\ \mathtt{e}_1' \arrow[r, phantom, "\mathcal{R}^\circ"] & \mathtt{e}_2' \end{tikzcd}$$

Note the use of the open extension $\mathcal{R}^\circ$. This means, for example, that if $\mathtt{e}_1\ \mathcal{R}\ \mathtt{e}_2$ we require that the move $\mathtt{e}_1 \xrightarrow{k?_B x} \mathtt{f}_1$ be matched by a move $\mathtt{e}_2 \xrightarrow{k?_B x} \mathtt{f}_2$ where $\mathtt{f}_2$ is such that for all values $\vdash \mathtt{v} : B$ we have $\mathtt{f}_1[\mathtt{v}/\mathtt{x}]\ \mathcal{R}\ \mathtt{f}_2[\mathtt{v}/\mathtt{x}]$. Thus in the terminology of (Milner, Parrow, and Walker 1992) our definition corresponds to the *late* version of bisimulation.

$\mathcal{R}$ is a *first-order strong bisimulation* iff $\mathcal{R}$ and $\mathcal{R}^{-1}$ are first-order strong simulations. Let $\sim^1$ be the largest first-order strong bisimulation.

**Proposition 1.** *$\sim^1$ is an equivalence.*

**Proof.** Use diagram chases to show that if $\mathcal{R}$ is a first-order strong simulation then so are $I$ and $\mathcal{R}\mathcal{R}$. The result follows. □

Unfortunately, $\sim^1$ is not a congruence for $\mu$CML$^+$, since we have:

$$\mathtt{add(1,-1)} \sim^1 \mathtt{add(-1,1)}$$

however, sending the thunked expressions on channel $\mathtt{a}$ we get:

$$\mathtt{transmit(a,fn\,x => add(1,-1))} \not\sim^1 \mathtt{transmit(a,fn\,x => add(-1,1))}$$

since the lhs can perform the move:

$$\mathtt{transmit(a,fn\,x => add(1,-1))} \xrightarrow{\mathtt{a!fn\,x => add(1,-1)}} \mathtt{()}$$



but this can only be matched by the rhs up to strong bisimulation:

$$\mathtt{transmit(a,fn\,x => add(-1,1))} \xrightarrow{\mathtt{a!fn\,x => add(-1,1)}} \mathtt{()}$$

The problem is that the definition of strong bisimulation demands that the actions performed by expressions match up to syntactic identity, rather than up to strong bisimulation. In fact, it is easy to verify that the only first-order strong bisimulation which is a congruence for $\mu$CML is the identity relation.

To find a satisfactory treatment of bisimulation for $\mu$CML, we need to look to *higher-order bisimulation*, where the structure of the labels is accounted for. To this end, given a closed type-indexed relation $\mathcal{R}$, define its *extension to labels* $\mathcal{R}^l$ as:

$$\frac{}{\tau\ \mathcal{R}^l_A\ \tau} \qquad \frac{\mathtt{v}\ \mathcal{R}_A\ \mathtt{w}}{\checkmark\mathtt{v}\ \mathcal{R}^l_A\ \checkmark\mathtt{w}} \qquad \frac{}{\mathtt{k?}_B\mathtt{x}\ \mathcal{R}^l_A\ \mathtt{k?}_B\mathtt{x}} \qquad \frac{\mathtt{v}\ \mathcal{R}_B\ \mathtt{w}}{\mathtt{k!}_B\mathtt{v}\ \mathcal{R}^l_A\ \mathtt{k!}_B\mathtt{w}}$$

Then $\mathcal{R}$ is a *higher-order strong simulation* iff it is structure preserving and the following diagram can be completed:

$$\begin{tikzcd} \mathtt{e}_1 \arrow[r, phantom, "\mathcal{R}"] \arrow[d, "l_1"'] & \mathtt{e}_2 \\ \mathtt{e}_1' & \end{tikzcd} \quad \text{as} \quad \begin{tikzcd} \mathtt{e}_1 \arrow[r, phantom, "\mathcal{R}"] \arrow[d, "l_1"'] & \mathtt{e}_2 \arrow[d, "l_2"'] \\ \mathtt{e}_1' \arrow[r, phantom, "\mathcal{R}^\circ"] & \mathtt{e}_2' \end{tikzcd} \text{ where } l_1\ \mathcal{R}^l\ l_2$$

Let $\sim^h$ be the largest higher-order strong bisimulation.

**Proposition 2.** *$\sim^h$ is a congruence.*

**Proof.** Use a similar technique to the proof of Proposition 1 to show that $\sim^h$ is an equivalence. To show that $\sim^h$ is a congruence, define $\mathcal{R}$ as:

$$\mathcal{R} = \{(C[e], C[f]) \mid e \sim^h f\}$$

and then show by induction on $C$ that $\mathcal{R}$ is a simulation. The result follows. □

For many purposes, strong bisimulation is too fine an equivalence as it is sensitive to the number of reductions performed by expressions. This means it will not even validate elementary properties such as $\beta$-reduction. We require the looser *weak bisimulation* which allows $\tau$ reductions to be ignored.

Let $\stackrel{\hat{l}}{\Longrightarrow}$ be $\Longrightarrow$ if $l = \tau$ and $\stackrel{l}{\Longrightarrow}$ otherwise. Then $\mathcal{R}$ is a *higher-order weak simulation* iff it is structure preserving and the following diagram can be completed:

$$\begin{array}{ccc} e_1 & \mathcal{R} & e_2 \\ {\scriptstyle l_1}\downarrow & & \\ e_1' & & \end{array} \quad \text{as} \quad \begin{array}{ccc} e_1 & \mathcal{R} & e_2 \\ {\scriptstyle l_1}\downarrow & & \Downarrow{\scriptstyle \hat{l}_2} \\ e_1' & \mathcal{R}^\circ & e_2' \end{array} \quad \text{where } l_1 \; \mathcal{R}^l \; l_2$$

A *higher-order weak bisimulation* is a higher-order weak simulation whose inverse is also a higher-order weak simulation. Let $\approx^h$ be the largest higher-order weak bisimulation.

**Proposition 3.** *$\approx^h$ is a congruence.*

**Proof.** Given in (Ferreira, Hennessy, and Jeffrey 1995), using a variant of Gordon's (1995) presentation Howe's (1989) proof technique. Note that this proof relies on the fact that we are considering the subset of $\mu$CML without `always`, and hence do not have to consider initial $\tau$-actions in summations, which present the same problems as in the first-order case (Milner 1989). □

Unfortunately, this equivalence does not have many pleasant mathematical properties. For example none of the usual equations for products are true:

$$\begin{aligned} \mathtt{fst\,(e\,,f)} &\not\approx^h \mathtt{e} \\ \mathtt{snd\,(e\,,f)} &\not\approx^h \mathtt{f} \\ \mathtt{(fst\,e\,,snd\,e)} &\not\approx^h \mathtt{e} \end{aligned}$$

(For each counter-example consider an expression with side-effects, such as `cell`.)

In the next section we shall consider a variant of $\mu$CML which uses a restrictive type system to provide more pleasant mathematical properties of programs. We shall then show a translation from $\mu$CML into the restricted language, which is correct up to weak bisimulation.

## 3 Concurrent monadic ML

In the previous section, we showed how to define an operational semantics for CML which can be used as the basis of a bisimulation equivalence between programs. Unfortunately, this equivalence does not have pleasant mathematical properties. For example $\beta$-conversion does not hold:

$$\mathtt{(fn\,x\,{=}{>}\,(x\,,x))(cell\,(a,b))} \not\approx^h \mathtt{(cell\,(a,b)\,,cell\,(a,b))}$$

Because CML computations are non-trivial (CML processes may diverge, and can have side-effects) we cannot use the standard mathematical models of typed $\lambda$-calculi such as cartesian closed categories (Lambek and Scott 1986).



In this section, we present a *Concurrent Monadic ML* (*CMML*) a variant of CML with a type system based on Moggi's (1991) computation types. Such type systems have proved popular in giving an elegant treatment to functional languages with non-trivial computation, such as the Haskell I/O system (Gordon et al. 1994).

CMML can be provided with an operational semantics similar to that given to CML in the previous section, although the semantics is much simpler, and has pleasant properties such as forming a category with finite products and a restricted class of exponentials.

The language presented here ($\mu$CMML) is a subset of the language presented in (Jeffrey 1995a).

### 3.1 Syntax

The main difference between CMML and CML is that the distinction between values and expressions is handled by the CMML type system rather than as a separate syntactic category. For example, in CML we have:

$$\vdash \mathtt{0} : \mathtt{int}\ \text{(a value)} \qquad \vdash \mathtt{add\,(\text{-}1\,,1)} : \mathtt{int}\ \text{(an expression)}$$

whereas in CMML we have:

$$\vdash 0 : \mathsf{int}\ \text{(an expression)} \qquad \vdash \mathsf{add}\langle -1, 1\rangle : \mathsf{int\,comp}\ \text{(an expression )}$$

This uses an explicit type constructor $A\,\mathsf{comp}$ to represent computations which return results of type $A$. For example $\mathsf{add}\langle -1, 1\rangle$ returns the result $0$, so it has the type $\mathsf{int\,comp}$.

Moggi (1991) proposed two syntactic constructions for manipulating computation types:

- the expression $[e]$ which immediately returns $e$, and
- the expression $\mathsf{let}\,x \Leftarrow e\,\mathsf{in}\,f$ which evaluates $e$, binds the result to $x$ and then evaluates $f$.

For example $(1+1)+(1+1)$ can be calculated as:

$$\begin{array}{l} \mathsf{let}\,x \Leftarrow [1] \\ \mathsf{in}\,\mathsf{let}\,y \Leftarrow \mathsf{add}\langle x, x\rangle \\ \quad \mathsf{in}\,\mathsf{add}\langle y, y\rangle \end{array}$$

Note that expressions written in $\mu$CMML tend to be more long-winded than their $\mu$CML equivalents: this is because the flow of execution through a $\mu$CMML program is made explicit by the use of $\mathsf{let}$-expressions. Such an explicit language may seem overly verbose to functional programmers used to programming in the SML style, where execution order is implicit in the left-to-right evaluation order. However, as we shall see, making execution order explicit has the benefit of a simpler semantics and better equational properties.

Using an explicit type constructor for computation has the advantage that the only terms which perform computation are those of type $A\,\mathsf{comp}$, and that an expression of any other type is guaranteed to be in normal form. This gives us the normal form results (Proposition 4 below):

- the only closed term of type $\mathsf{unit}$ is $()$,
- the only closed terms of type $\mathsf{bool}$ are $\mathsf{true}$ and $\mathsf{false}$,
- the only closed terms of type $\mathsf{int}$ are $\ldots, -1, 0, 1, \ldots$,
- the only closed terms of type $\mathsf{chan}$ are $\mathsf{a}, \mathsf{b}, \ldots$,
- the only closed terms of type $A * B$ are of the form $\langle e, f \rangle$, and
- the only closed terms of type $A \to B\,\mathsf{comp}$ are of the form $\mathsf{rec}\,x = \mathsf{fn}\,y \Rightarrow e$.

These results make the operational semantics much simpler to define, for example rather than two rules for function application:

$$\frac{e \xrightarrow{\alpha} e'}{e f \xrightarrow{\alpha} e' f} \qquad \frac{e \xrightarrow{\checkmark v} e'}{e f \xrightarrow{\tau} e' \,\|\, \mathtt{let}\, y = \mathtt{f}\ \mathtt{in}\, g[v/x]}[v = \mathtt{rec}\ x = \mathtt{fn}\ y \mathtt{=>} g]$$

we only need one simple β-reduction rule:

$$\frac{}{e f \xrightarrow{\tau} g[f/y][e/x]}[e = (\mathsf{rec}\,x = \mathsf{fn}\,y \Rightarrow g)]$$

The simplicity of the operational semantics rests on the normal form result described above, but this requires a somewhat non-standard treatment of projections on pairs. In $\mu$CML projections are given using `fst` and `snd`, for example a function to swap a pair is:

$$\vdash \mathtt{fn}\ x \mathtt{=>} (\mathtt{snd}\ x, \mathtt{fst}\ x) : A * B \mathtt{->} B * A$$

If we were to allow `fst` and `snd` in CMML we would no longer have the normal form result described above. However, projections on pairs are useful both practically and as the categorical basis of products. In CMML we use a restricted form of projections which maintains the normal form result: we use Pascal-style record field selection on *lvalues* rather than ML-style selection functions. If $x$ is a variable of type $A * B$ then $x.\mathsf{l}$ is an expression of type $A$, and $x.\mathsf{r}$ is an expression of type $B$. For example a CMML function to swap a pair is:

$$\vdash \mathsf{fn}\,x \Rightarrow [\langle x.\mathsf{r}, x.\mathsf{l} \rangle] : A * B \to (B * A)\,\mathsf{comp}$$

Similarly, we need to use a restricted form of function space, since the result of any function application should be a computation. This means that rather than the CML function type:

$$\frac{\Gamma, x : A \mathtt{->} B, y : A \vdash e : B}{\Gamma \vdash \mathtt{rec}\ x = \mathtt{fn}\ y \mathtt{=>} e : A \mathtt{->} B} \qquad \frac{\Gamma \vdash e : A \mathtt{->} B \quad \Gamma \vdash f : A}{\Gamma \vdash e f : B}$$



we have the restricted CMML function type:

$$\frac{\Gamma, x : A \to B\,\mathsf{comp}, y : A \vdash e : B\,\mathsf{comp}}{\Gamma \vdash \mathsf{rec}\,x = \mathsf{fn}\,y \Rightarrow e : A \to B\,\mathsf{comp}} \qquad \frac{\Gamma \vdash e : A \to B\,\mathsf{comp} \quad \Gamma \vdash f : A}{\Gamma \vdash e f : B\,\mathsf{comp}}$$

For example there is no CMML projection function with type $A * B \to A$, instead we have:

$$\vdash \mathsf{fn}\,x \Rightarrow [x.\mathsf{l}] : A * B \to A\,\mathsf{comp}$$

The concurrent features of $\mu$CMML are similar to those of $\mu$CML$^+$, for example a concurrent communication is given by:

$$k!0 \,\|\, k? \xrightarrow{\tau} [()] \,\|\, [0]$$

We will now give the grammar and type system for $\mu$CMML.

Integers and channels are given as for $\mu$CML:

$$n ::= \cdots \mid -1 \mid 0 \mid 1 \mid \cdots$$
$$k ::= \mathsf{a} \mid \mathsf{b} \mid \cdots$$

Basic functions are given by the grammar:

$$c ::= \mathsf{add} \mid \mathsf{mul} \mid \mathsf{leq}$$

Expressions are given by the grammar:

$$\begin{aligned} e ::= {} & \mathsf{true} \mid \mathsf{false} \mid n \mid k \mid () \mid \mathsf{rec}\,x = \mathsf{fn}\,x \Rightarrow e \mid c\,e \\ & \mid \mathsf{if}\,e\,\mathsf{then}\,e\,\mathsf{else}\,e \mid \mathsf{let}\,x \Leftarrow e\,\mathsf{in}\,e \mid e\,e \mid lv \mid [e] \mid \langle e, e \rangle \\ & \mid \delta \mid e \,\|\, e \mid e \,\Box\, e \mid e!_A e \mid e?_A \end{aligned}$$

Lvalues are given by the grammar:

$$lv ::= x \mid lv.\mathsf{l} \mid lv.\mathsf{r}$$

Types are given by the grammar:

$$A ::= \mathsf{unit} \mid \mathsf{bool} \mid \mathsf{int} \mid \mathsf{chan} \mid A * A \mid A \to A\,\mathsf{comp} \mid A\,\mathsf{comp}$$

Typing is given by Tables 10 and 11.

**Proposition 4.** *We have the following normal form results:*

1. *If* $\Gamma \vdash e : \mathsf{unit}$ *then* $e$ *is an lvalue or* $e = ()$.
2. *If* $\Gamma \vdash e : \mathsf{bool}$ *then* $e$ *is an lvalue or* $e = \mathsf{true}$ *or* $e = \mathsf{false}$.
3. *If* $\Gamma \vdash e : \mathsf{int}$ *then* $e$ *is an lvalue or* $e = n$.
4. *If* $\Gamma \vdash e : \mathsf{chan}$ *then* $e$ *is an lvalue or* $e = k$.
5. *If* $\Gamma \vdash e : A * B$ *then* $e$ *is an lvalue or* $e = \langle f, g \rangle$.
6. *If* $\Gamma \vdash e : A \to B\,\mathsf{comp}$ *then* $e$ *is an lvalue or* $e = (\mathsf{rec}\,x = \mathsf{fn}\,y \Rightarrow f)$.

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool}} \quad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool}} \quad \frac{}{\Gamma \vdash n : \mathsf{int}} \quad \frac{}{\Gamma \vdash k : \mathsf{chan}} \quad \frac{}{\Gamma \vdash () : \mathsf{unit}}$$

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash c\, e : B\,\mathsf{comp}}[c : A \to B\,\mathsf{comp}] \quad \frac{\Gamma, x : A \to B\,\mathsf{comp}, y : A \vdash e : B\,\mathsf{comp}}{\Gamma \vdash \mathsf{rec}\, x = \mathsf{fn}\, y \Rightarrow e : A \to B\,\mathsf{comp}}$$

$$\frac{\Gamma \vdash e : \mathsf{bool} \quad \Gamma \vdash f : A\,\mathsf{comp} \quad \Gamma \vdash g : A\,\mathsf{comp}}{\Gamma \vdash \mathsf{if}\, e\, \mathsf{then}\, f\, \mathsf{else}\, g : A\,\mathsf{comp}}$$

$$\frac{\Gamma \vdash e : A\,\mathsf{comp} \quad \Gamma, x : A \vdash f : B\,\mathsf{comp}}{\Gamma \vdash \mathsf{let}\, x \Leftarrow e\, \mathsf{in}\, f : B\,\mathsf{comp}} \quad \frac{\Gamma \vdash e : A \to B\,\mathsf{comp} \quad \Gamma \vdash f : A}{\Gamma \vdash e\, f : B\,\mathsf{comp}}$$

$$\frac{}{\Gamma, x : A \vdash x : A} \quad \frac{\Gamma \vdash x : A}{\Gamma, y : B \vdash x : A}[x \neq y] \quad \frac{\Gamma \vdash lv : A * B}{\Gamma \vdash lv.\mathsf{l} : A} \quad \frac{\Gamma \vdash lv : A * B}{\Gamma \vdash lv.\mathsf{r} : B}$$

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash [e] : A\,\mathsf{comp}} \quad \frac{\Gamma \vdash e : A \quad \Gamma \vdash f : B}{\Gamma \vdash \langle e, f \rangle : A * B}$$

$$\frac{}{\Gamma \vdash \delta : A\,\mathsf{comp}} \quad \frac{\Gamma \vdash e : A\,\mathsf{comp} \quad \Gamma \vdash f : B\,\mathsf{comp}}{\Gamma \vdash e \| f : B\,\mathsf{comp}}$$

$$\frac{\Gamma \vdash e : A\,\mathsf{comp} \quad \Gamma \vdash f : A\,\mathsf{comp}}{\Gamma \vdash e \Box f : A\,\mathsf{comp}} \quad \frac{\Gamma \vdash e : \mathsf{chan} \quad \Gamma \vdash f : A}{\Gamma \vdash e!_A f : \mathsf{unit}\,\mathsf{comp}} \quad \frac{\Gamma \vdash e : \mathsf{chan}}{\Gamma \vdash e?_A : A\,\mathsf{comp}}$$

TABLE 10. Types for $\mu$CMML expressions

$$\begin{aligned} \mathsf{add} &: \mathsf{int} * \mathsf{int} \to \mathsf{int}\,\mathsf{comp} \\ \mathsf{mul} &: \mathsf{int} * \mathsf{int} \to \mathsf{int}\,\mathsf{comp} \\ \mathsf{leq} &: \mathsf{int} * \mathsf{int} \to \mathsf{bool}\,\mathsf{comp} \end{aligned}$$

TABLE 11. Types for $\mu$CMML basic functions



**Proof.** A case analysis on the proof of $\Gamma \vdash e : A$. □

When $\Gamma \vdash e : A$ and $\Gamma, x : A \vdash f : B$, define the substitution $\Gamma \vdash f[e/x] : B$ as normal, except that:

$$lv.\mathsf{l}[e/x] = \pi(lv[e/x]) \qquad lv.\mathsf{r}[e/x] = \pi'(lv[e/x])$$

where:

$$\pi\langle e, f\rangle = e \quad \pi lv = lv.\mathsf{l} \qquad \pi'\langle e, f\rangle = f \quad \pi' lv = lv.\mathsf{r}$$

Note that this is well-defined because of Proposition 4.5.

As an example $\mu$CMML program, consider a one-place buffer:

$$\begin{aligned} \mathsf{cell}_A &: \mathsf{chan} * \mathsf{chan} \to B\,\mathsf{comp} \\ \mathsf{cell}_A\langle i, o\rangle &\stackrel{\text{def}}{=} \mathsf{let}\, x \Leftarrow i?_A\, \mathsf{in}\, \mathsf{let}\, y \Leftarrow o!_A x\, \mathsf{in}\, \mathsf{cell}_A\langle i, o\rangle \end{aligned}$$

Comparing this definition with its $\mu$CML equivalent is instructive, so we shall repeat the definition here:

$$\begin{aligned} \mathtt{cell}_A &: \mathtt{chan * chan \text{->} } B \\ \mathtt{cell}_A\,\mathtt{(x, y)} &\stackrel{\text{def}}{=} \mathtt{cell}_A\,\mathtt{(snd\,(send}_A\,\mathtt{(y, accept}_A\,\mathtt{x), (x, y)))} \end{aligned}$$

Writing programs in $\mu$CMML can be repetitive, because of the number of let-expressions required. However, the let-expressions are precisely what controls the flow of execution through a $\mu$CMML program, so it is easier to recognize the behaviour of a $\mu$CMML program. In the above example, it requires some thought to realize that $\mathtt{cell}_A(\mathtt{a}, \mathtt{b})$ will input on $\mathtt{a}$ before outputting the result on $\mathtt{b}$, and that the process does not just simply diverge, whereas the execution of the $\mu$CMML equivalent is much more obvious.

In Section 4 we shall see that $\mu$CML programs can be translated into $\mu$CMML, and that in particular we can perform some simple equational reasoning to transform `cell` into $\mathsf{cell}$.

### *3.2 Operational semantics*

The operational semantics for $\mu$CMML is given in Tables 12–15. It is similar to that of $\mu$CML, except that it is simpler, due to the normal form results in Proposition 4. For example, since any closed term of type $\mathsf{bool}$ must be either $\mathsf{true}$ or $\mathsf{false}$, the only two rules required for if-statements in $\mu$CMML are:

$$\frac{}{\mathsf{if}\,\mathsf{true}\,\mathsf{then}\, f\, \mathsf{else}\, g \xrightarrow{\tau} f} \quad \frac{}{\mathsf{if}\,\mathsf{false}\,\mathsf{then}\, f\, \mathsf{else}\, g \xrightarrow{\tau} g}$$

This can be compared with the more complex three rules required for $\mu$CML:

$$\frac{\mathtt{e} \xrightarrow{\sqrt{\mathtt{true}}} \mathtt{e}'}{\mathtt{if\ e\ then\ f\ else\ } g \xrightarrow{\tau} \mathtt{e}' \| \mathtt{f}} \quad \frac{\mathtt{e} \xrightarrow{\sqrt{\mathtt{false}}} \mathtt{e}'}{\mathtt{if\ e\ then\ f\ else\ } g \xrightarrow{\tau} \mathtt{e}' \| g}$$

$$\frac{\mathtt{e} \xrightarrow{\alpha} \mathtt{e}'}{\mathtt{if\ e\ then\ f\ else\ } g \xrightarrow{\alpha} \mathtt{if\ e'\ then\ f\ else\ } g}$$

$$\frac{e \xrightarrow{\alpha} e'}{\mathsf{let}\, x \Leftarrow e\, \mathsf{in}\, f \xrightarrow{\alpha} \mathsf{let}\, x \Leftarrow e'\, \mathsf{in}\, f}$$

$$\frac{e \xrightarrow{\alpha} e'}{e \,\|\, f \xrightarrow{\alpha} e' \,\|\, f} \qquad \frac{f \xrightarrow{l} f'}{e \,\|\, f \xrightarrow{l} e \,\|\, f'} \qquad \frac{e \xrightarrow{\tau} e'}{e \,\Box\, f \xrightarrow{\tau} e' \,\Box\, f} \qquad \frac{f \xrightarrow{\tau} f'}{e \,\Box\, f \xrightarrow{\tau} e \,\Box\, f'}$$

TABLE 12. CMML operational semantics: static rules

In the operational semantics of $\mu$CML, terms in many contexts can reduce, whereas there are far fewer reduction contexts in $\mu$CMML. In fact, looking at the sequential sub-language of $\mu$CMML (without $\|$ or $\Box$) the only reduction context is $\mathsf{let}$:

$$\frac{e \xrightarrow{\alpha} e'}{\mathsf{let}\, x \Leftarrow e\, \mathsf{in}\, f \xrightarrow{\alpha} \mathsf{let}\, x \Leftarrow e'\, \mathsf{in}\, f}$$

Many of the operational rules in $\mu$CML require spawning off concurrent processes, whereas in $\mu$CMML the main rule which produces extra concurrent processes is $\beta$-reduction for $\mathsf{let}$-expressions:

$$\frac{e \xrightarrow{\surd g} e'}{\mathsf{let}\, x \Leftarrow e\, \mathsf{in}\, f \xrightarrow{\tau} e' \,\|\, f[g/x]}$$

The other significant difference between the operational semantics for $\mu$CML and $\mu$CMML is the treatment of summation. In $\mu$CML choice is only allowed between guarded expressions $ge_1 \oplus ge_2$, whereas in $\mu$CMML choice is allowed between arbitrary expressions $e \,\Box\, f$. In particular, this means we need operational rules for when processes in a choice can perform silent reductions:

$$\frac{e \xrightarrow{\tau} e'}{e \,\Box\, f \xrightarrow{\tau} e' \,\Box\, f} \qquad \frac{f \xrightarrow{\tau} f'}{e \,\Box\, f \xrightarrow{\tau} e \,\Box\, f'}$$

and when processes in a choice can return a value:

$$\frac{e \xrightarrow{\surd g} e'}{e \,\Box\, f \xrightarrow{\tau} e' \,\|\, [g]} \qquad \frac{f \xrightarrow{\surd g} f'}{e \,\Box\, f \xrightarrow{\tau} f' \,\|\, [g]}$$

Note that we are using rules for choice based on CSP (Hoare 1985) external choice rather than CCS (Milner 1989) summation. This is because we will be using $\approx^h$ as our equivalence on programs, and CCS summation does not preserve weak bisimulation. We have used slightly different termination rules for choice from CSP, in order to ensure *forward commutativity* of the resulting transition system (see Section 3.3 below for why this is important).

As an example of an $\mu$CMML program execution, one possible run of the one-place buffer is given in Table 16, which can be compared to the equivalent $\mu$CML execution in Table 9. The extra complexity of the $\mu$CML execution is due to the book-keeping work that $\mu$CML has to do because an expression of any type has the capability of computation, so the operational semantics has to



$$\frac{e \xrightarrow{\mu} e'}{e \,\Box\, f \xrightarrow{\mu} e'} \qquad \frac{f \xrightarrow{\mu} f'}{e \,\Box\, f \xrightarrow{\mu} f'}$$

TABLE 13. CMML operational semantics: dynamic rules

$$\frac{}{e\, f \xrightarrow{\tau} g[f/y][e/x]}[e = (\mathsf{rec}\, x = \mathsf{fn}\, y \Rightarrow g)] \qquad \frac{}{c\, e \xrightarrow{\tau} [\delta(c, e)]}$$

$$\frac{}{\mathsf{if}\,\mathsf{true}\,\mathsf{then}\, f\, \mathsf{else}\, g \xrightarrow{\tau} f} \qquad \frac{}{\mathsf{if}\,\mathsf{false}\,\mathsf{then}\, f\, \mathsf{else}\, g \xrightarrow{\tau} g}$$

$$\frac{e \xrightarrow{\surd g} e'}{\mathsf{let}\, x \Leftarrow e\, \mathsf{in}\, f \xrightarrow{\tau} e' \,\|\, f[g/x]}$$

$$\frac{e \xrightarrow{k!_A g} e' \quad f \xrightarrow{k?_A x} f'}{e \,\|\, f \xrightarrow{\tau} e' \,\|\, f'[g/x]} \qquad \frac{e \xrightarrow{k?_A x} e' \quad f \xrightarrow{k!_A g} f'}{e \,\|\, f \xrightarrow{\tau} e'[g/x] \,\|\, f'}$$

$$\frac{e \xrightarrow{\surd g} e'}{e \,\Box\, f \xrightarrow{\tau} e' \,\|\, [g]} \qquad \frac{f \xrightarrow{\surd g} f'}{e \,\Box\, f \xrightarrow{\tau} f' \,\|\, [g]}$$

TABLE 14. CMML operational semantics: silent reductions

$$\frac{}{[e] \xrightarrow{\surd e} \delta} \qquad \frac{}{k!_A e \xrightarrow{k!_A e} [()]} \qquad \frac{}{k?_A \xrightarrow{k?_A x} [x]}$$

TABLE 15. CMML operational semantics: axioms

$$\begin{aligned}
&\mathsf{cell}\langle i, o\rangle \\
&\quad \xrightarrow{\tau} \mathsf{let}\, x \Leftarrow i?\, \mathsf{in}\, \mathsf{let}\, y \Leftarrow o!x\, \mathsf{in}\, \mathsf{cell}\langle i, o\rangle \\
&\quad \xrightarrow{i?e} \mathsf{let}\, x \Leftarrow [e]\, \mathsf{in}\, \mathsf{let}\, y \Leftarrow o!x\, \mathsf{in}\, \mathsf{cell}\langle i, o\rangle \\
&\quad \xrightarrow{\tau} \mathsf{let}\, y \Leftarrow o!e\, \mathsf{in}\, \mathsf{cell}\langle i, o\rangle \\
&\quad \xrightarrow{o!e} \mathsf{let}\, y \Leftarrow [()]\, \mathsf{in}\, \mathsf{cell}\langle i, o\rangle \\
&\quad \xrightarrow{\tau} \mathsf{cell}\langle i, o\rangle
\end{aligned}$$

TABLE 16. CMML operational semantics: example reduction

allow computation at any point in evaluation. For example, in the evaluation of $\mathtt{send}\,(e\,,f\,)$, both $e$ and $f$ have to terminate before the communication can happen, so if $e \stackrel{\surd k}{\Longrightarrow} \delta$ and $f \stackrel{\surd v}{\Longrightarrow} \delta$ then:

$$
\begin{array}{l}
\mathtt{send}\,(e\,,f\,) \\
\quad \xrightarrow{\tau} \mathtt{let}\ x = (e\,,f\,)\ \mathtt{in}\ \mathtt{sync}\,(\mathtt{transmit}\ x\,) \\
\quad \Longrightarrow \mathtt{let}\ x = \mathtt{let}\ y = f\ \mathtt{in}\,\langle k\,,y\rangle\ \mathtt{in}\ \mathtt{sync}\,(\mathtt{transmit}\ x\,) \\
\quad \Longrightarrow \mathtt{let}\ x = \langle k\,,v\rangle\ \mathtt{in}\ \mathtt{sync}\,(\mathtt{transmit}\ x\,) \\
\quad \longrightarrow \mathtt{sync}\,(\mathtt{transmit}\langle k\,,v\rangle) \\
\quad \longrightarrow \mathtt{sync}\,[k\,!\,v] \\
\quad \longrightarrow k\,!\,v \\
\quad \xrightarrow{k\,!\,v} (\,)
\end{array}
$$

whereas the type system for $\mu$CMML ensures that $e$ and $f$ do not have to be evaluated before $e!f$ can communicate.

### *3.3 Bisimulation*

We can define 'structure-preserving' and 'bisimulation' for $\mu$CMML in the same way as for $\mu$CML.

**Proposition 5.** $\approx^h$ *is a congruence for* $\mu$CMML.

**Proof.** Similar to the proof of Proposition 3. □

In comparison to $\mu$CML, this equivalence has some pleasant mathematical properties. In particular we can define a category of $\mu$CML terms, where:

- objects are $\mu$CML types,
- morphisms from $A$ to $B$ are expressions with one free variable $x : A \vdash e : B$ viewed up to higher-order weak bisimulation $\approx^{h^\circ}$,
- the identity morphism is $x : A \vdash x : A$, and
- morphism composition is substitution: $(x : A \vdash e : B);(y : B \vdash f : C)$ is $x : A \vdash f[e/y] : C$.

This category has binary products $A * B$ with projections:

$$x : A * B \vdash x.\mathsf{l} : A \qquad x : A * B \vdash x.\mathsf{r} : B$$

and mediating morphism:

$$\frac{x : A \vdash e : B \quad x : A \vdash f : C}{x : A \vdash \langle e, f\rangle : B * C}$$

To verify that these satisfy the defining property for products we have to show that (whenever $\Gamma \vdash g : A * B$):

$$\pi\langle e, f\rangle \approx^h e$$



$$\pi'\langle e, f\rangle \approx^h f$$
$$g \approx^h \langle \pi g\,, \pi' g\rangle$$

The category has an initial object unit with mediating morphism:

$$x : A \vdash () : \mathsf{unit}$$

since (whenever $\Gamma \vdash e : \mathsf{unit}$):

$$e \approx^h ()$$

The category has monad given by the $\_\mathsf{comp}$ type constructor with action on morphisms given by:

$$\frac{x : A \vdash e : B}{y : A\,\mathsf{comp} \vdash \mathsf{let}\, x \Leftarrow y\, \mathsf{in}\, [e] : B\,\mathsf{comp}}$$

and strict monadic structure given by natural transformations:

$$x : A \vdash [x] : A\,\mathsf{comp}$$
$$x : A\,\mathsf{comp}\,\mathsf{comp} \vdash \mathsf{let}\, y \Leftarrow x\, \mathsf{in}\, y : A\,\mathsf{comp}$$
$$x : A * (B\,\mathsf{comp}) \vdash \mathsf{let}\, y \Leftarrow x.\mathsf{r}\, \mathsf{in}\, [\langle x.\mathsf{l}, y\rangle] : (A * B)\,\mathsf{comp}$$

since (whenever $\Gamma \vdash e : A\,\mathsf{comp}$, $\Gamma, x : A \vdash f : B\,\mathsf{comp}$, $\Gamma, y : B \vdash g : C\,\mathsf{comp}$ and $x, y \notin \Gamma$):

$$\mathsf{let}\, x \Leftarrow [e]\, \mathsf{in}\, f \approx^h f[e/x]$$
$$\mathsf{let}\, x \Leftarrow e\, \mathsf{in}\, [x] \approx^h e$$
$$\mathsf{let}\, y \Leftarrow \mathsf{let}\, x \Leftarrow e\, \mathsf{in}\, f\, \mathsf{in}\, g \approx^h \mathsf{let}\, x \Leftarrow e\, \mathsf{in}\, \mathsf{let}\, y \Leftarrow f\, \mathsf{in}\, g$$

This category has all $\_\mathsf{comp}$ exponentials given by $A \rightarrow B\,\mathsf{comp}$ with the currying adjunction given by:

$$\frac{x : A * B \vdash e : C\,\mathsf{comp}}{y : A \vdash \mathsf{fn}\, z \Rightarrow \mathsf{let}\, x \Leftarrow [\langle y, z\rangle]\, \mathsf{in}\, e : B \rightarrow C\,\mathsf{comp}}$$

$$\frac{x : A \vdash e : B \rightarrow C\,\mathsf{comp}}{y : A * B \vdash \mathsf{let}\, x \Leftarrow [y.\mathsf{l}]\, \mathsf{in}\, e(y.\mathsf{r}) : C}$$

since (whenever $\Gamma, x : A \vdash e : B\,\mathsf{comp}$, $\Gamma \vdash f : A$ and $\Gamma \vdash g : A \rightarrow B\,\mathsf{comp}$):

$$(\mathsf{fn}\, x \Rightarrow e)f \approx^h e[x/f]$$
$$\mathsf{fn}\, x \Rightarrow (g\, x) \approx^h g$$

The categorical structure of $\mu$CMML is based on Moggi's (1991) general theory of computation types, and is discussed further in (Jeffrey 1995a; Jeffrey 1995b).

In order to prove the above bisimulations, we need to show some properties about the labelled transition systems produced by $\mu$CMML programs. In particular we require the lts to be *value deterministic*:

$$\text{if} \quad \begin{tikzcd} e \arrow[r, "\surd f"] \arrow[d, "\surd g"'] & e' \\ e'' & \end{tikzcd} \quad \text{then } f = g \text{ and } e' = e''$$

*single-valued*:

$$\text{if } e \xrightarrow{\surd f} e' \xrightarrow{l} e'' \text{ then } l \neq \surd g$$

*forward commutative*:

$$\text{if} \quad \begin{tikzcd} e \arrow[r, "\surd f"] \arrow[d, "\alpha"'] & e' \\ e'' & \end{tikzcd} \quad \text{then} \quad \begin{tikzcd} e \arrow[r, "\surd f"] \arrow[d, "\alpha"'] & e' \arrow[d, "\alpha"'] \\ e'' \arrow[r, "\surd f^{\circ}"] & e''' \end{tikzcd}$$

and *backward commutative*:

$$\text{if} \quad \begin{tikzcd} e \arrow[r, "\surd f"] & e' \arrow[d, "\alpha"'] \\ & e''' \end{tikzcd} \quad \text{then} \quad \begin{tikzcd} e \arrow[r, "\surd f"] \arrow[d, "\alpha"'] & e' \arrow[d, "\alpha"'] \\ e'' \arrow[r, "\surd f^{\circ}"] & e''' \end{tikzcd}$$

From these properties we can show that:

$$\text{if } e \xrightarrow{\surd f} e' \text{ then } e \approx^h e' \,\|\, [f]$$

which is used in proving the above bisimulations.

## 4 Translating CML to CMML

As we have seen, the operational semantics for $\mu$CML is more complex than that of $\mu$CMML, since terms of any type can reduce. However, in this section we shall show that there is a translation from $\mu$CML$^+$ into $\mu$CMML, and that the translation is correct up to weak bisimulation.

### *4.1 The translation*

This translation is based on Moggi's (1991) translation of the call-by-value λ-calculus into the computational λ-calculus.



$$\begin{aligned} T[\![\texttt{bool}]\!] &= \mathsf{bool} \\ T[\![\texttt{chan}]\!] &= \mathsf{chan} \\ T[\![\texttt{int}]\!] &= \mathsf{int} \\ T[\![\texttt{unit}]\!] &= \mathsf{unit} \\ T[\![A * B]\!] &= T[\![A]\!] * T[\![B]\!] \\ T[\![A \texttt{ -> } B]\!] &= T[\![A]\!] \rightarrow T[\![B]\!]\,\mathsf{comp} \\ T[\![A \texttt{ event}]\!] &= T[\![A]\!]\,\mathsf{comp} \end{aligned}$$

TABLE 17. Translation of $\mu$CML$^+$ types into $\mu$CML

First, we translate each $\mu$CML$^+$ type $A$ into an $\mu$CMML type $T[\![A]\!]$. The only tricky question is how to translate the function space $A \texttt{ -> } B$. Moggi has proposed $A\,\mathsf{comp} \rightarrow B\,\mathsf{comp}$ for the *call-by-name* translation (where functions take computations as arguments) and $A \rightarrow B\,\mathsf{comp}$ for the *call-by-value* translation (where functions take canonical forms as arguments). Since $\mu$CML is a call-by-value language, we shall use the latter translation. This is given in Table 17, and can be extended to contexts:

$$T[\![x_1 : A_1, \ldots, x_n : A_n]\!] = x_1 : T[\![A_1]\!], \ldots, x_n : T[\![A_n]\!]$$

The trick for translating $\mu$CML$^+$ terms into $\mu$CMML terms is to provide two translations:

- translate $\mu$CML$^+$ values $\Gamma \vdash v : A$
  into $\mu$CMML expressions $T[\![\Gamma]\!] \vdash V[\![v]\!] : T[\![A]\!]$, and
- translate $\mu$CML$^+$ expressions $\Gamma \vdash e : A$
  into $\mu$CMML computations $T[\![\Gamma]\!] \vdash E[\![e]\!] : T[\![A]\!]\,\mathsf{comp}$.

This reflects the intuition that any expression in $\mu$CML$^+$ can perform computation, whereas in $\mu$CMML only terms of type $A\,\mathsf{comp}$ can compute. The two translations are given in Tables 18 and 19.

Note that most of the $\mu$CML$^+$ expressions have the same form, which is to evaluate their argument in a let-expression before continuing. This corresponds to the notion that $\mu$CML$^+$ is a call-by-value language, where expressions are evaluated to canonical form before being manipulated.

For example, the translation of `cell` is given in Table 20, where to save space we have used the fact that:

$$\begin{aligned} E[\![\texttt{send}_A\, e]\!] &\approx^h \mathsf{let}\, x \Leftarrow E[\![e]\!]\, \mathsf{in}\, x.\mathsf{l}!x.\mathsf{r} \\ E[\![\texttt{accept}_A\, e]\!] &\approx^h \mathsf{let}\, x \Leftarrow E[\![e]\!]\, \mathsf{in}\, x? \end{aligned}$$

This translation is almost unreadable, and very inefficient, but we can use β-

$$
\begin{aligned}
V[\![\mathtt{true}]\!] &= \mathsf{true}\\
V[\![\mathtt{false}]\!] &= \mathsf{false}\\
V[\![\mathtt{n}]\!] &= n\\
V[\![\mathtt{k}]\!] &= k\\
V[\![\mathtt{()}]\!] &= ()\\
V[\![\langle v, w\rangle]\!] &= \langle V[\![v]\!], V[\![w]\!]\rangle\\
V[\![\mathtt{rec}\,x\,\mathtt{=}\,\mathtt{fn}\,y\,\mathtt{=>}\,e]\!] &= \mathsf{rec}\,x = \mathsf{fn}\,y \Rightarrow E[\![e]\!]\\
V[\![x]\!] &= x\\
V[\![\mathtt{[}ge\mathtt{]}]\!] &= E[\![ge]\!]
\end{aligned}
$$

TABLE 18. Translation of $\mu$CML$^+$ values into $\mu$CML

reduction to remove some extraneous $\mathsf{let}$s:

$$
\begin{aligned}
V[\![\mathtt{cell}]\!] \approx^h\ & \mathsf{rec}\,x_1 = \mathsf{fn}\,x_2 \Rightarrow\\
& \mathsf{let}\,x_4 \Leftarrow \mathsf{let}\,x_5 \Leftarrow \mathsf{let}\,x_6 \Leftarrow \mathsf{let}\,x_8 \Leftarrow \mathsf{let}\,x_{10} \Leftarrow x_2.\mathsf{l}?\,\mathsf{in}\,[\langle x_2.\mathsf{r}, x_{10}\rangle]\\
& \qquad\qquad\qquad\quad \mathsf{in}\,x_8.\mathsf{l}!x_8.\mathsf{r}\\
& \qquad\qquad\quad \mathsf{in}\,[\langle x_6, x_2\rangle]\\
& \qquad \mathsf{in}\,[x_5.\mathsf{r}]\\
& \mathsf{in}\,x_1\,x_4
\end{aligned}
$$

Then associativity gives:

$$
\begin{aligned}
V[\![\mathtt{cell}]\!] \approx^h\ & \mathsf{rec}\,x_1 = \mathsf{fn}\,x_2 \Rightarrow\\
& \mathsf{let}\,x_{10} \Leftarrow x_2.\mathsf{l}?\\
& \mathsf{in}\,\mathsf{let}\,x_8 \Leftarrow [\langle x_2.\mathsf{r}, x_{10}\rangle]\\
& \quad \mathsf{in}\,\mathsf{let}\,x_6 \Leftarrow x_8.\mathsf{l}!x_8.\mathsf{r}\\
& \qquad \mathsf{in}\,\mathsf{let}\,x_5 \Leftarrow [\langle x_6, x_2\rangle]\\
& \qquad\quad \mathsf{in}\,\mathsf{let}\,x_4 \Leftarrow [x_5.\mathsf{r}]\,\mathsf{in}\,x_1\,x_4
\end{aligned}
$$

So further use of $\beta$-reduction gives:

$$
\begin{aligned}
V[\![\mathtt{cell}]\!] \approx^h\ & \mathsf{rec}\,x_1 = \mathsf{fn}\,x_2 \Rightarrow\\
& \mathsf{let}\,x_{10} \Leftarrow x_2.\mathsf{l}?\\
& \mathsf{in}\,\mathsf{let}\,x_6 \Leftarrow x_2.\mathsf{r}!x_{10}\\
& \quad \mathsf{in}\,x_1\,x_2
\end{aligned}
$$

and since (up to $\alpha$-conversion) this is the definition of $\mathsf{cell}$, we have:

$$V[\![\mathtt{cell}]\!] \approx^h \mathsf{cell}$$

This example shows that it is easy to perform syntactic manipulations on $\mu$CMML expressions to drastically reduce them in size, and improve their effi-



$$
\begin{aligned}
E[\![v]\!] &= [V[\![v]\!]]\\
E[\![\mathtt{fst}\,e]\!] &= \mathsf{let}\,x \Leftarrow E[\![e]\!]\,\mathsf{in}\,[x.\mathsf{l}]\\
E[\![\mathtt{snd}\,e]\!] &= \mathsf{let}\,x \Leftarrow E[\![e]\!]\,\mathsf{in}\,[x.\mathsf{r}]\\
E[\![\mathtt{add}\,e]\!] &= \mathsf{let}\,x \Leftarrow E[\![e]\!]\,\mathsf{in}\,\mathsf{add}\,x\\
E[\![\mathtt{mul}\,e]\!] &= \mathsf{let}\,x \Leftarrow E[\![e]\!]\,\mathsf{in}\,\mathsf{mul}\,x\\
E[\![\mathtt{leq}\,e]\!] &= \mathsf{let}\,x \Leftarrow E[\![e]\!]\,\mathsf{in}\,\mathsf{leq}\,x\\
E[\![\mathtt{transmit}_A\,e]\!] &= \mathsf{let}\,x \Leftarrow E[\![e]\!]\,\mathsf{in}\,[x.\mathsf{l}!_{T[\![A]\!]}x.\mathsf{r}]\\
E[\![\mathtt{receive}_A\,e]\!] &= \mathsf{let}\,x \Leftarrow E[\![e]\!]\,\mathsf{in}\,[x?_{T[\![A]\!]}]\\
E[\![\mathtt{choose}\,e]\!] &= \mathsf{let}\,x \Leftarrow E[\![e]\!]\,\mathsf{in}\,[x.\mathsf{l} \,\square\, x.\mathsf{r}]\\
E[\![\mathtt{spawn}\,e]\!] &= \mathsf{let}\,x \Leftarrow E[\![e]\!]\,\mathsf{in}\,x\,() \parallel [()]\\
E[\![\mathtt{sync}\,e]\!] &= \mathsf{let}\,x \Leftarrow E[\![e]\!]\,\mathsf{in}\,x\\
E[\![\mathtt{wrap}\,e]\!] &= \mathsf{let}\,x \Leftarrow E[\![e]\!]\,\mathsf{in}\,[\mathsf{let}\,y \Leftarrow x.\mathsf{l}\,\mathsf{in}\,x.\mathsf{r}\,y]\\
E[\![\mathtt{never}\,e]\!] &= \mathsf{let}\,x \Leftarrow E[\![e]\!]\,\mathsf{in}\,[\delta]\\
E[\![\mathtt{if}\,e\,\mathtt{then}\,f\,\mathtt{else}\,g]\!] &= \mathsf{let}\,x \Leftarrow E[\![e]\!]\,\mathsf{in}\,\mathsf{if}\,x\,\mathsf{then}\,E[\![f]\!]\,\mathsf{else}\,E[\![g]\!]\\
E[\![\mathtt{(}e\mathtt{,}f\mathtt{)}]\!] &= \mathsf{let}\,x \Leftarrow E[\![e]\!]\,\mathsf{in}\,\mathsf{let}\,y \Leftarrow E[\![f]\!]\,\mathsf{in}\,[\langle x, y\rangle]\\
E[\![\mathtt{let}\,x\,\mathtt{=}\,e\,\mathtt{in}\,f]\!] &= \mathsf{let}\,x \Leftarrow E[\![e]\!]\,\mathsf{in}\,E[\![f]\!]\\
E[\![ef]\!] &= \mathsf{let}\,x \Leftarrow E[\![e]\!]\,\mathsf{in}\,\mathsf{let}\,y \Leftarrow E[\![f]\!]\,\mathsf{in}\,x\,y\\
E[\![e \parallel f]\!] &= E[\![e]\!] \parallel E[\![f]\!]\\
E[\![v?_A]\!] &= V[\![v]\!]?_{T[\![A]\!]}\\
E[\![v!_A w]\!] &= V[\![v]\!]!_{T[\![A]\!]}V[\![w]\!]\\
E[\![\delta]\!] &= \delta\\
E[\![ge_1 \oplus ge_2]\!] &= E[\![ge_1]\!] \,\square\, E[\![ge_2]\!]\\
E[\![ge \Rightarrow v]\!] &= \mathsf{let}\,x \Leftarrow E[\![ge]\!]\,\mathsf{in}\,V[\![v]\!]x
\end{aligned}
$$

TABLE 19. Translation of $\mu$CML$^+$ expressions into $\mu$CML

$$
\begin{array}{l}
V[\![\texttt{cell}]\!] \approx^h \mathsf{rec}\,x_1 = \mathsf{fn}\,x_2 \Rightarrow \\
\qquad \mathsf{let}\,x_3 \Leftarrow [x_1] \\
\qquad \mathsf{in}\,\mathsf{let}\,x_4 \Leftarrow \mathsf{let}\,x_5 \Leftarrow \mathsf{let}\,x_6 \Leftarrow \mathsf{let}\,x_8 \Leftarrow \mathsf{let}\,x_9 \Leftarrow \mathsf{let}\,x_{11} \Leftarrow [x_2] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{in}\,[x_{11}.\mathsf{r}] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{in}\,\mathsf{let}\,x_{10} \Leftarrow \mathsf{let}\,x_{12} \Leftarrow \mathsf{let}\,x_{13} \Leftarrow [x_2] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{in}\,[x_{13}.\mathsf{l}] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{in}\,x_{12}? \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{in}\,[\langle x_9, x_{10}\rangle] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{in}\,x_8.\mathsf{l}!x_8.\mathsf{r} \\
\qquad\qquad\qquad\qquad\qquad \mathsf{in}\,\mathsf{let}\,x_7 \Leftarrow [x_2] \\
\qquad\qquad\qquad\qquad\qquad\quad \mathsf{in}\,[\langle x_6, x_7\rangle] \\
\qquad\qquad\qquad \mathsf{in}\,[x_5.\mathsf{r}] \\
\qquad\quad \mathsf{in}\,x_3\,x_4
\end{array}
$$

TABLE 20. Example translation of $\mu$CML$^+$ into $\mu$CMML

ciency. This suggests that $\mu$CMML may be a suitable virtual machine language for a $\mu$CML compiler, where verifiable peephole optimizations can be performed.

### 4.2 *Correctness of the translation*

We will now show that the translation of $\mu$CML$^+$ into $\mu$CMML is correct up to bisimulation. We will do this by defining an appropriate notion of weak bisimulation between $\mu$CML and $\mu$CMML programs. This proof uses Milner and Sangiorgi's (1992) technique of 'bisimulation up to'.

A *closed type-indexed relation between* $\mu$CML *and* $\mu$CMML is a family of relations:

$$
\begin{array}{l}
\mathcal{R}^e_A \subseteq \{(\mathtt{e}, e) \mid \vdash \mathtt{e} : A, \vdash e : T[\![A]\!]\,\mathsf{comp}\} \\
\mathcal{R}^v_A \subseteq \{(\mathtt{v}, e) \mid \vdash \mathtt{v} : A, \vdash e : T[\![A]\!]\}
\end{array}
$$

For any closed type-indexed relation $\mathcal{R}$, let its *open extension* $\mathcal{R}^{e\circ}$ be defined as:

$$\mathtt{e}\ \mathcal{R}^{e\circ}_{\vec{x}:\vec{A},B}\ e \text{ iff } \mathtt{e}[\vec{\mathtt{v}}/\vec{x}]\ \mathcal{R}^e_B\ e[V[\![\vec{\mathtt{v}}]\!]/\vec{x}] \text{ for all } \vdash \vec{\mathtt{v}} : \vec{A}.$$

A closed type-indexed relation $\mathcal{R}$ is structure-preserving iff:

- if $\mathtt{v}\ \mathcal{R}^v_A\ e$ and $A$ is a base type then $\mathtt{v} = e$,
- if $\langle \mathtt{v}_1, \mathtt{v}_2\rangle\ \mathcal{R}^v_{A_1 * A_2}\ \langle e_1, e_2\rangle$ then $\mathtt{v}_i\ \mathcal{R}^v_{A_i}\ e_i$,
- if $\mathtt{[ge]}\ \mathcal{R}^v_{A\,\mathtt{event}}\ e$ then $\mathtt{ge}\ \mathcal{R}^e_A\ e$, and
- if $\mathtt{v}\ \mathcal{R}^v_{A\mathtt{->}B}\ e$ then for all $\vdash \mathtt{w} : A$ we have $\mathtt{v}\,\mathtt{w}\ \mathcal{R}^e_B\ e(V[\![\mathtt{w}]\!])$.



A closed type-indexed relation can be extended to labels as:

$$
\frac{}{\tau\ \mathcal{R}^l_A\ \tau}
\qquad
\frac{\mathtt{v}\ \mathcal{R}^v_A\ e}{\checkmark\,\mathtt{v}\ \mathcal{R}^l_A\ \checkmark\,e}
\qquad
\frac{}{\mathtt{k}?_B\mathtt{x}\ \mathcal{R}^l_A\ k?_{T[\![B]\!]}x}
\qquad
\frac{\mathtt{v}\ \mathcal{R}^v_B\ e}{\mathtt{k}!_B\mathtt{v}\ \mathcal{R}^l_A\ k!_{T[\![B]\!]}e}
$$

A closed type-indexed relation between $\mu$CML and $\mu$CMML is a *higher-order weak bisimulation* iff it is structure preserving and we can complete the following diagrams:

$$
\begin{tikzcd}
\mathtt{e}_1 \arrow[r, phantom, "\mathcal{R}^e"] \arrow[d, "l_1"'] & e_2 \\
\mathtt{e}_1' &
\end{tikzcd}
\quad \text{as} \quad
\begin{tikzcd}
\mathtt{e}_1 \arrow[r, phantom, "\mathcal{R}^e"] \arrow[d, "l_1"'] & e_2 \arrow[d, Rightarrow, "\hat{l}_2"'] \\
\mathtt{e}_1' \arrow[r, phantom, "\mathcal{R}^{e\circ}"] & e_2'
\end{tikzcd}
\quad \text{where } l_1\ \mathcal{R}^l\ l_2
$$

and:

$$
\begin{tikzcd}
\mathtt{e}_1 \arrow[r, phantom, "\mathcal{R}^e"] & e_2 \arrow[d, "l_2"'] \\
& e_2'
\end{tikzcd}
\quad \text{as} \quad
\begin{tikzcd}
\mathtt{e}_1 \arrow[r, phantom, "\mathcal{R}^e"] \arrow[d, Rightarrow, "\hat{l}_1"'] & e_2 \arrow[d, "l_2"'] \\
\mathtt{e}_1' \arrow[r, phantom, "\mathcal{R}^{e\circ}"] & e_2'
\end{tikzcd}
\quad \text{where } l_1\ \mathcal{R}^l\ l_2
$$

A closed type-indexed relation between $\mu$CML and $\mu$CMML is a *higher-order strong bisimulation up to* $(\leq, \sqsubseteq)$ iff it is structure preserving and we can complete the following diagrams:

$$
\begin{tikzcd}
\mathtt{e}_1 \arrow[r, phantom, "\mathcal{R}^e"] \arrow[d, "l_1"'] & e_2 \\
\mathtt{e}_1' &
\end{tikzcd}
\quad \text{as} \quad
\begin{tikzcd}
\mathtt{e}_1 \arrow[r, phantom, "\mathcal{R}^e"] \arrow[d, "l_1"'] & e_2 \arrow[d, "l_2"'] \\
\mathtt{e}_1' \arrow[r, phantom, "\leq\mathcal{R}^{e\circ}\sqsubseteq"] & e_2'
\end{tikzcd}
\quad \text{where } l_1\ \mathcal{R}^l\ l_2
$$

and:

$$
\begin{tikzcd}
\mathtt{e}_1 \arrow[r, phantom, "\mathcal{R}^e"] & e_2 \arrow[d, "l_2"'] \\
& e_2'
\end{tikzcd}
\quad \text{as} \quad
\begin{tikzcd}
\mathtt{e}_1 \arrow[r, phantom, "\mathcal{R}^e"] \arrow[d, "l_1"'] & e_2 \arrow[d, "l_2"'] \\
\mathtt{e}_1' \arrow[r, phantom, "\leq\mathcal{R}^{e\circ}\sqsubseteq"] & e_2'
\end{tikzcd}
\quad \text{where } l_1\ \mathcal{R}^l\ l_2
$$

An *expansion* on $\mu$CMML (and similarly on $\mu$CML) is a weak bisimulation $\mathcal{R}$

such that the following diagrams can be completed:

$$\begin{tikzcd} e_1 \arrow[d, "l_1"'] & \mathcal{R} & e_2 \\ e_1' & & \end{tikzcd} \quad \text{as} \quad \begin{tikzcd} e_1 \arrow[d, "l_1"'] & \mathcal{R} & e_2 \arrow[d, Rightarrow, "l_2"'] \\ e_1' & \mathcal{R}^\circ & e_2' \end{tikzcd} \quad \text{where } l_1 \mathrel{\mathcal{R}^l} l_2$$

and:

$$\begin{tikzcd} e_1 & \mathcal{R} & e_2 \arrow[d, "l_2"'] \\ & & e_2' \end{tikzcd} \quad \text{as} \quad \begin{tikzcd} e_1 \arrow[d, "\hat{l}_1"'] & \mathcal{R} & e_2 \arrow[d, "l_2"'] \\ e_1' & \mathcal{R}^\circ & e_2' \end{tikzcd} \quad \text{where } l_1 \mathrel{\mathcal{R}^l} l_2$$

Let $\lesssim$ be the largest expansion.

**Proposition 6.** *$\lesssim$ is a precongruence on $\mu$CML and $\mu$CMML.*

**Proof.** Similar to Proposition 3. □

For example, the preorder $\leq_\beta$ given by $\beta$-reducing in all contexts is an expansion:

$$\frac{}{e\,f \geq_\beta g[f/y][e/x]}[e = (\mathsf{rec}\,x = \mathsf{fn}\,y \Rightarrow g)] \qquad \frac{}{\mathsf{let}\,x \Leftarrow [e]\,\mathsf{in}\,f \geq_\beta f[e/x]}$$

$$\frac{}{\mathsf{if\,true\,then}\,f\,\mathsf{else}\,g \geq_\beta f} \qquad \frac{}{\mathsf{if\,false\,then}\,f\,\mathsf{else}\,g \geq_\beta g}$$

$$\frac{}{e \geq_\beta e} \qquad \frac{e \geq_\beta f \geq_\beta g}{e \geq_\beta g} \qquad \frac{e \geq_\beta f}{C[e] \geq_\beta C[f]}$$

**Proposition 7.** *If $e \leq_\beta f$ then $e \lesssim f$.*

**Proof.** Show that each of the axioms forms an expansion. The result then follows from Proposition 6. □

We can use the proof technique of strong bisimulation up to $(\leq, \sqsubseteq)$ to show that the translation from $\mu$CML to $\mu$CMML forms a weak bisimulation.

**Proposition 8.** *Any strong bisimulation up to $(\gtrsim, \lesssim)$ is a weak bisimulation.*

**Proof.** An adaptation of the results in (Sangiorgi and Milner 1992). □

**Proposition 9.** *The translation of $\mu$CML$^+$ into $\mu$CMML is a strong bisimulation up to $(\geq_\beta, \leq_\beta)$.*

**Proof.** Let $\mathcal{R}$ be:

$$\mathcal{R}^e_A = \{(e, E\llbracket e \rrbracket) \mid \vdash e : A\} \qquad \mathcal{R}^v_A = \{(v, V\llbracket v \rrbracket) \mid \vdash v : A\}$$



and let $L\llbracket l \rrbracket$ be the extension of the translation to labels:

$$L\llbracket \tau \rrbracket = \tau \qquad L\llbracket \checkmark v \rrbracket = \checkmark V\llbracket v \rrbracket$$
$$L\llbracket k!_A v \rrbracket = k!_{T\llbracket A \rrbracket} V\llbracket v \rrbracket \qquad L\llbracket k?_A x \rrbracket = k?_{T\llbracket A \rrbracket} x$$

First show that the translation respects substitution of values, that is:

$$E\llbracket (e[v/x]) \rrbracket = E\llbracket e \rrbracket [V\llbracket v \rrbracket / x]$$

Next show by induction on $ge$ that if $ge \xrightarrow{l} e$ then $l$ is an input or output label.

Then show that for any $\vdash e : A$, if $e \xrightarrow{l} e'$ then $E\llbracket e \rrbracket \xrightarrow{L\llbracket l \rrbracket} \geq_\beta E\llbracket f' \rrbracket$ and $e' \geq_\beta f'$. This is an induction on the proof of reduction, for example if:

$$\frac{ge \xrightarrow{\alpha} e'}{ge \Rightarrow v \xrightarrow{\alpha} ve'}$$

where $v = \mathtt{rec}\,y = \mathtt{fn}\,z \Rightarrow g$ then by induction:

$$E\llbracket ge \rrbracket \xrightarrow{L\llbracket \alpha \rrbracket} \geq_\beta E\llbracket f' \rrbracket \qquad e' \geq_\beta f'$$

and so:

$$\begin{aligned} & E\llbracket ge \Rightarrow v \rrbracket \\ = \; & \mathsf{let}\,x \Leftarrow E\llbracket ge \rrbracket\,\mathsf{in}\,V\llbracket v \rrbracket\,x \\ \xrightarrow{L\llbracket \alpha \rrbracket} \geq_\beta \; & \mathsf{let}\,x \Leftarrow E\llbracket f' \rrbracket\,\mathsf{in}\,V\llbracket v \rrbracket\,x \\ \geq_\beta \; & \mathsf{let}\,x \Leftarrow E\llbracket f' \rrbracket\,\mathsf{in}\,E\llbracket g \rrbracket [x/z][V\llbracket v \rrbracket / y] \\ = \; & E\llbracket \mathtt{let}\,z = f'\,\mathtt{in}\,g[v/y] \rrbracket \end{aligned}$$

and:

$$\begin{aligned} & ve' \\ & \geq_\beta vf' \\ & \geq_\beta \mathtt{let}\,z = f'\,\mathtt{in}\,g[v/y] \end{aligned}$$

The other cases are similar.

Then show that for any $\vdash e : A$, if $E\llbracket e \rrbracket \xrightarrow{l_2} e'$ then $e \xrightarrow{l_1} \geq_\beta e'$, $L\llbracket l_1 \rrbracket = l_2$ and $e' \geq_\beta E\llbracket e' \rrbracket$. This is an induction on $e$, for example if:

$$\frac{E\llbracket ge \rrbracket \xrightarrow{\alpha_1} e'}{E\llbracket ge \Rightarrow v \rrbracket \xrightarrow{\alpha_2} \mathsf{let}\,x \Leftarrow e'\,\mathsf{in}\,V\llbracket v \rrbracket\,x}$$

where $v = \mathtt{rec}\,y = \mathtt{fn}\,z \Rightarrow g$ then by induction:

$$ge \xrightarrow{\alpha_1} \geq_\beta e' \qquad L\llbracket \alpha_1 \rrbracket = \alpha_2 \qquad e' \geq_\beta E\llbracket e' \rrbracket$$

and so:

$$\begin{aligned} & ge \Rightarrow v \\ & \xrightarrow{\alpha_1} \geq_\beta ve' \\ & \geq_\beta \mathtt{let}\,z = e'\,\mathtt{in}\,g[v/y] \end{aligned}$$

and:

$$\begin{aligned}&\mathsf{let}\, x \Leftarrow e' \,\mathsf{in}\, V[\![v]\!]\, x\\ &\quad \geq_\beta \mathsf{let}\, x \Leftarrow E[\![\mathtt{e}']\!] \,\mathsf{in}\, V[\![v]\!]\, x\\ &\quad \geq_\beta \mathsf{let}\, x \Leftarrow E[\![\mathtt{e}']\!] \,\mathsf{in}\, E[\![g]\!][x/z][V[\![v]\!]/y]\\ &\quad = E[\![\mathtt{let}\, z = \mathtt{e}' \,\mathtt{in}\, g[v/y]]\!]\end{aligned}$$

The other cases are similar. □

**Proposition 10.** *$\mathtt{e}$ is weakly bisimilar to $E[\![\mathtt{e}]\!]$.*

**Proof.** Follows from Propositions 7, 8 and 9. □

It follows from this that weak bisimulation for $\mu$CMML is at least as fine as weak bisimulation for $\mu$CML$^+$.

**Proposition 11.** *If $E[\![\mathtt{e}]\!] \approx^h E[\![\mathtt{f}]\!]$ then $\mathtt{e} \approx^h \mathtt{f}$.*

**Proof.** Follows immediately from Proposition 10. □

However, note that the translation is *not* necessarily fully abstract, in that we have not shown that this implication is an 'if and only if'. This is because the bisimulation is higher-order, and the clause for bisimulation between functions requires the functions to agree on all arguments, not just ones which are the image of $E[\![\_]\!]$.

## References


Ferreira, W., M. Hennessy, and A. Jeffrey (1995). A theory of weak bisimulation for core CML. COGS Comp. Sci. Tech. Report 05/95, Univ. Sussex.

Gordon, A. (1995). Bisimilarity as a theory of functional programming. In *Proc. MFPS 95*, Number 1 in Electronic Notes in Comp. Sci. Springer-Verlag.

Gordon, A. et al. (1994). 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.

Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Prentice-Hall.

Howe, D. (1989). Equality in lazy computation systems. In *Proc. LICS 89*, pp. 198–203.

Jeffrey, A. (1995a). A fully abstract semantics for a concurrent functional language with monadic types. In *Proc. LICS 95*, pp. 255–264.

Jeffrey, A. (1995b). A fully abstract semantics for a nondeterministic functional language with monadic types. In *Proc. MFPS 95*, Electronic Notes in Comput. Sci. Elsevier.





Lambek, J. and P. J. Scott (1986). *Introduction to Higher Order Categorical Logic*. Cambridge University Press.

Milner, R. (1989). *Communication and Concurrency*. Prentice-Hall.

Milner, R., J. Parrow, and D. Walker (1992). A calculus of mobile proceses. *Inform. and Comput. 100*(1), 1–77.

Milner, R., M. Tofte, and R. Harper (1990). *The Definition of Standard ML*. MIT Press.

Moggi, E. (1991). Notions of computation and monad. *Inform. and Comput. 93*, 55–92.

Reppy, J. (1991). A higher-order concurrent langauge. In *Proc. SIGPLAN 91*, pp. 294–305.

Reppy, J. (1992). *Higher-Order Concurrency*. Ph.D thesis, Cornell Univ.

Sangiorgi, D. and R. Milner (1992). Techniques of 'weak bisimulation up to'. In *Proc. CONCUR 92*. Springer Verlag. LNCS 630.