UNIVERSITY OF SUSSEX

COMPUTER SCIENCE

# A Theory of Weak Bisimulation for Core CML

William Ferreira
Matthew Hennessy
Alan Jeffrey

Report 05/95 September 1995

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

ISSN 1350–3170

## A Theory of Weak Bisimulation for Core CML

WILLIAM FERREIRA, MATTHEW HENNESSY and ALAN JEFFREY

ABSTRACT. Concurrent ML is an extension of Standard ML of New Jersey with concurrent features similar to those of process algebra. Reppy has given it an operational semantics based on reductions of configurations, using entire programs rather than program fragments. The existing semantics are not, therefore, compositional, and do not support compositional reasoning (for example equational reasoning about program fragments).

In this paper, we present a compositional operational semantics for a fragment of CML, based on higher-order process algebra, and use this to define weak bisimulation for CML. We give some small examples of proofs about CML expressions, and show that our semantics corresponds to Reppy's up to weak first-order bisimulation.

### 1 Introduction

There have been various attempts to extend standard programming languages with concurrent or distributed features, [10, 16, 25]. Concurrent ML (CML) [28, 30] is a practical and elegant example. The language Standard ML is extended with two new type constructors, one for generating communication channels, and the other for *delayed computations*. By adding to the language a small number of constants to manipulate objects of these new types a new language is obtained which combines the functional features of ML with the communication capabilities of CCS, [20]. Thus the language has all the functional and higher-order features of ML but programs also have the ability to spawn new *computation threads* and these independent threads can communicate with each other by transmitting values along communication channels. It has been implemented and a formal semantics has been given for a significant subset, [5, 30]. As Reppy pointed out in [29], "Another useful direction would be to build a 'theory' of CML programs to allow reasoning about their correctness." The purpose of this paper is to provide one such theory.

In [5, 30] an operational semantics is given for a language called $\lambda_{cv}$. This may be viewed as a concurrent version of the call-by-value $\lambda$-calculus of Plotkin, [27] but is also contains many of the interesting features of CML. Indeed it may be viewed as an extension of a *mini*-CML, similar to what what we will call in this paper $\mu$CML, as it contains the core elements of CML; the extension is obtained by adding new constructs, which do not appear in CML but which facilitate the description of the operational semantics. This operational semantics


This work is carried out in the context of EC BRA 7166 CONCUR 2.
William Ferreira was funded by a CASE studentship from British Telecom.

is given in terms of a reduction relation between *configurations*, multi-sets of $\lambda_{cv}$ closed expressions or programs. Unfortunately this operational semantics is not compositional, in that the behaviour of a $\lambda_{cv}$ expression, or indeed configuration, is not determined by that of its constituents.

Here we give a compositional operational semantics in terms of a labelled transition system for $\mu$CML programs. This not only describes the evaluation steps of programs, as in [30], but also their communication potentials, in terms of their ability to input and output values along communication channels.

We then proceed to demonstrate the usefulness of this compositional operational semantics by using it to define a version of *weak observational equivalence*, [20], suitable for $\mu$CML. We prove that, modulo the usual problems associated with the choice operator of CCS, our chosen equivalence is preserved by all $\mu$CML contexts and therefore may be used as the basis for reasoning about CML programs. In this paper we do not investigate in detail the resulting theory but confine ourselves to pointing out some of its salient features; for example standard identities one would expect of a call-by-value $\lambda$-calculus are given and we also show that certain algebraic laws common to process algebras, [20], hold.

We now explain in more detail the contents of the remainder of the paper.

IN SECTION 2 we describe the language $\mu$CML, a subset of CML. It is a typed language, with base types for channel names, booleans and integers, and type constructors for pairs, functions and delayed computations; these last are called Event types. It has the standard constructs and constants associated with the base types and with pairs and functions. In addition it has a selection of the CML constructs and constants for manipulating delayed computations; spawn generates a new computation thread, sync launches a delayed computation, transmit and receive construct basic delayed computations for sending and receiving values, while wrap is used to combine delayed computations. In short we focus on much the same subset of CML as [30]; the major omission is that $\mu$CML has no facility for generating new channel names. This is for convenience only; we believe that our semantics can be extended to handle channel generation, using techniques common to the $\pi$-calculus, [21, 22, 31], but this would obscure much of our exposition.

This section then proceeds with an exposition of our operational semantics, in terms of a labelled transition system. In order to describe all possible states which can arise during the computation of a well-typed $\mu$CML program we need to extend the language. This extension is twofold. The first consists in adding the constants of event type used by Reppy in [30] to define $\lambda_{cv}$, i.e. constants to denote certain delayed computations. This extended language, which we call $\mu\text{CML}^{cv}$, essentially coincides with the $\lambda_{cv}$, the language used in [30], except for the omissions cited above. However to obtain a compositional semantics we



make further extensions to $\mu\text{CML}^{cv}$. We add a parallel operator $\|$, commonly used in process algebras, which allows us to use programs in place of the multisets of programs of [30]. The final addition is more subtle; we allow expressions which correspond to the synced versions of Reppy's constants. Thus the labelled transition system uses as states programs from a language which we call $\mu\text{CML}^{+}$. This language is a superset of $\mu\text{CML}^{cv}$, which is our version of Reppy's $\lambda_{cv}$, which in turn is a superset of $\mu$CML, our mini-version of CML.

IN SECTION 3 we discuss semantic equivalences defined on the labelled transition of Section 2. We demonstrate the inadequacies of the obvious adaptations of *strong* and *weak* bisimulation equivalence, [20], and then consider adaptations of *higher-order* and *irreflexive* bisimulations from [32]. Finally we suggest a new variation called *hereditary* bisimulation equivalence which overcomes some of the problems encountered with using higher-order and irreflexive bisimulations.

IN SECTION 4 we show that hereditary bisimulation is preserved by all $\mu$CML contexts. This is an application of the proof method originally suggested in [17] but the proof is further complicated by the fact that hereditary bisimulations are defined in terms of pairs of relations satisfying mutually dependent properties.

IN SECTION 5 we briefly discuss the resulting algebraic theory of $\mu$CML expressions. This paper is intended only to lay the foundations of this theory and so here we simply indicate that our theory extends both that of call-by-value $\lambda$-calculus [27] and process algebras [20].

IN SECTION 6 we show that, up to weak bisimulation equivalence, our semantics coincides with the reduction semantics for $\lambda_{cv}$ presented in [30]. This technical result applies only to the common sub-language, namely $\mu\text{CML}^{cv}$.

IN SECTION 7 we briefly consider other approaches to the semantics of CML and related languages and we end with some suggestions for further work.

## 2 The Language

In this section we introduce our language $\mu$CML, a subset of Concurrent ML [28, 30]. We describe the syntax, including a typing system, and an operational semantics in terms of a labelled transition system.

The type expressions for our language are given by:

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

Thus we have four base types, unit, chan, bool and int; the latter two are simply examples of useful base types and one could easily include more. These types are closed under three constructors, pairing, function space, and the less common event type constructor. Our language may be viewed as a typed $\lambda$-calculus augmented with the type constructor $A\,\mathsf{event}$ for constructing *delayed computations* of type $A$.

Let *Chan* be a set of channel names ranged over by $k, k'$ etc. and let *Var* denote a set of variables ranged over by $x, y, \ldots$ The expressions of $\mu$CML are given by the following abstract syntax:

$$\begin{aligned}
e, f, g \in Exp &::= v \mid c\,e \mid \mathsf{if}\,e\,\mathsf{then}\,e\,\mathsf{else}\,e \mid (e, e) \mid \mathsf{let}\,x = e\,\mathsf{in}\,e \mid e\,e \\
v, w \in Val &::= l \mid \mathsf{fix}(x = \mathsf{fn}\,y \Rightarrow e) \mid x \\
c \in Const &::= \mathsf{fst} \mid \mathsf{snd} \mid \mathsf{add} \mid \mathsf{mul} \mid \mathsf{leq} \mid \mathsf{transmit}_A \mid \mathsf{receive}_A \\
&\quad\;\; \mid \mathsf{choose} \mid \mathsf{spawn} \mid \mathsf{sync} \mid \mathsf{wrap} \mid \mathsf{never} \mid \mathsf{always} \\
l \in Lit &::= \mathsf{true} \mid \mathsf{false} \mid k \mid () \mid 0 \mid 1 \mid \cdots
\end{aligned}$$

The main syntactic category is that of *Exp* which look very much like the set of expressions for an applied *call-by-value* version of the λ-calculus. There are the usual pairing and branching constructors, and three forms of application; the application of one expression to another, $e\,e$, the application of a constant to an expression, $c\,e$ and $\mathsf{let}\,x = e_1\,\mathsf{in}\,e_2$ representing the application to $e_1$ of the functional abstraction of $e_2$ over $x$. There is also a syntactic category of expressions of a particular form, called *Val*; these represent the objects to which functions may be applied and which also may be sent and received between computation threads. They are very restricted in form; either a predefined value for the base types, called *Lit*, or a recursively defined function, $\mathsf{fix}(x = \mathsf{fn}\,y \Rightarrow e)$. We will abbreviate this to $\mathsf{fn}\,y \Rightarrow e$ when $x$ does not occur in $e$.

Finally there are a small collection of constant functions. These consist of a representative sample of constants for manipulating objects of base type, $\mathsf{add}, \mathsf{mul}, \mathsf{leq}$, which could easily be extended, the projection functions $\mathsf{fst}$ and $\mathsf{snd}$, together with the set of constants for manipulating *delayed computations* taken directly from [30]:

- $\mathsf{transmit}$ and $\mathsf{receive}$, for constructing delayed computations which can send and receive values,
- $\mathsf{choose}$, for constructing alternatives between delayed computations,
- $\mathsf{spawn}$, for spawning new computational threads,
- $\mathsf{sync}$, for launching delayed computations,
- $\mathsf{wrap}$, for combining delayed computations,
- $\mathsf{never}$, for a delayed computation which always deadlocks, and
- $\mathsf{always}$, for a delayed computation which immediately terminates with a value.

Note that there is no method for generating channel names other than using the predefined set of names *Chan*.



There are two constructs in the language which bind occurrences of variables, $\mathsf{let}\,x = e_1\,\mathsf{in}\,e_2$ where free occurrences of $x$ in $e_2$ are bound and $\mathsf{fix}(x = \mathsf{fn}\,y \Rightarrow e)$ where free occurrences of both $x$ and $y$ in $e$ are bound. We will not dwell on the precise definitions of free and bound variables but simply use $fv(e)$ to denote the set of variables which have free occurrences in $e$. If $fv(e) = \emptyset$ then $e$ is said to be a *closed* expression, which we sometimes refer to as a *program*. We also use the standard notation of $e[v/x]$ to denote the substitution of the value $v$ for all free occurrences of $x$ in $e$ where bound names may be changed in order to avoid the capture of free variables in $v$.

We now examine briefly the type system for this language. The types for the constant functions of the language are given in Figure 1a; this is in agreement with the typing rules given in [30] for $\lambda_{cv}$. The constants $\mathsf{add}, \mathsf{mul}, \mathsf{spawn}$ have constant types associated with them, as have $\mathsf{transmit}_A$ and $\mathsf{receive}_A$; but the type of the latter pair is determined by the type subscript $A$. The type associated with the remaining constants should be interpreted polymorphically. Thus, for example $\mathsf{choose}$ has the type $A\,\mathsf{event} * A\,\mathsf{event} \to A\,\mathsf{event}$ for every type $A$.

This assignment of types to constant functions is used to infer types for arbitrary expressions in the standard way, using a type inference system. A *typing judgement* $\Gamma \vdash e : A$ consists of a *type assignment* $\Gamma$, an expression $e$ and a type $A$ such that $fv(e) \subseteq \{x_1, \ldots, x_n\}$. A *type assignment* is a sequence of the form $x_1 : t_1, \ldots, x_n : t_n$, where each $t_i$ is a type. Intuitively a type assignment should be read as "in the type assignment $\Gamma$ the expression $e$ has type $A$". The type inference system is given in Figure 1b and is straightforward. There are two structural rules, literals are assigned their natural types while the types of functional values are inferred using a minor modification of the standard rule for functional abstractions. The remaining constructs are also handled using standard inference rules, [12].

We now turn our attention to the operational semantics. In [30, 5] a reduction semantics is given to $\lambda_{cv}$ and since $\mu\mathrm{CML}^{cv}$ is a subset of $\lambda_{cv}$, this induces a reduction semantics for $\mu\mathrm{CML}^{cv}$; this is discussed in full in Section 6. The judgements in this reduction semantics are of the form:

$$C \xrightarrow{\tau} C'$$

where $C, C'$ are configurations which combine a closed expression with a runtime environment necessary for its evaluation. However this semantics is not compositional as the reductions of an expression can not be deduced directly from the reductions of it constituent components. Here we give a compositional operational semantics with four kinds of judgements:

- $e \xrightarrow{\tau} e'$, representing a one step evaluation or reduction,
- $e \xrightarrow{\sqrt{v}} e'$, representing the production of the value $v$, with a side effect $e'$,

$$
\begin{array}{rlrl}
\mathsf{fst} : & A * B \to A & \mathsf{transmit}_A : & \mathsf{chan} * A \to \mathsf{unit}\,\mathsf{event} \\
\mathsf{snd} : & A * B \to B & \mathsf{receive}_A : & \mathsf{chan} \to A\,\mathsf{event} \\
\mathsf{add} : & \mathsf{int} * \mathsf{int} \to \mathsf{int} & \mathsf{choose} : & A\,\mathsf{event} * A\,\mathsf{event} \to A\,\mathsf{event} \\
\mathsf{mul} : & \mathsf{int} * \mathsf{int} \to \mathsf{int} & \mathsf{spawn} : & (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit} \\
\mathsf{leq} : & \mathsf{int} * \mathsf{int} \to \mathsf{bool} & \mathsf{wrap} : & A\,\mathsf{event} * (A \to B) \to B\,\mathsf{event} \\
\mathsf{sync} : & A\,\mathsf{event} \to A & \mathsf{never} : & \mathsf{unit} \to A\,\mathsf{event} \\
\mathsf{always} : & A \to A\,\mathsf{event} & &
\end{array}
$$

FIGURE 1A. Type rules for $\mu$CML constant functions

$$
\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool}} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool}} \qquad \frac{}{\Gamma \vdash k : \mathsf{chan}}
$$

$$
\frac{}{\Gamma \vdash () : \mathsf{unit}} \qquad \frac{}{\Gamma \vdash n : \mathsf{int}} \qquad \frac{}{\Gamma, x : A \vdash x : A}
$$

$$
\frac{\Gamma, x : A \to B, y : A \vdash e : B}{\Gamma \vdash \mathsf{fix}(x = \mathsf{fn}\, y \Rightarrow e) : A \to B} \qquad \frac{\Gamma \vdash y : B}{\Gamma, x : A \vdash y : B}[x \neq y]
$$

$$
\frac{\Gamma \vdash e : A}{\Gamma \vdash c\, e : B}[c : A \to B] \qquad \frac{\Gamma \vdash e : A \to B \quad \Gamma \vdash f : A}{\Gamma \vdash e\, f : B} \qquad \frac{\Gamma \vdash e : A \quad \Gamma \vdash f : B}{\Gamma \vdash (e, f) : A * B}
$$

$$
\frac{\Gamma \vdash e : \mathsf{bool} \quad \Gamma \vdash f : A \quad \Gamma \vdash g : A}{\Gamma \vdash \mathsf{if}\, e\, \mathsf{then}\, f\, \mathsf{else}\, g : A} \qquad \frac{\Gamma \vdash e : A \quad \Gamma, x : A \vdash f : B}{\Gamma \vdash \mathsf{let}\, x = e\, \mathsf{in}\, f : B}
$$

FIGURE 1B. Type rules for $\mu$CML expressions.

- $e \xrightarrow{k?x} e'$, representing the potential to input a value $x$ along the channel $k$, and
- $e \xrightarrow{k!v} e'$, representing the output of the value $v$ along the channel $k$.

These are formally defined in Figure 2, but we first give an informal overview. In order to define these relations we introduce extra syntactic constructs. These are introduced as required in the overview but at the end of the section we give a formal definition of the extended language.

The rules for one step evaluation or reduction have much in common with those for a standard call-by-value λ-calculus. But in addition a closed expression $e$ of type $A$ should evaluate to a value of type $A$ and it is this production of values which is the subject of the second kind of judgement. However $\mu$CML is a language with side-effects and in particular the production of values can have side-effects and therefore instead of the simple judgement $e \xrightarrow{\surd} v$ we require the more complicated $e \xrightarrow{\surd v} e'$ where $e'$ represents the side-effect of the production of $v$. One example of the side-effects occurs in the evaluation of the $\mathsf{spawn}$ construct. Intuitively the closed expression $\mathsf{spawn}\, e$ always produces the trivial value $()$ but if $e$ is a function such as $\mathsf{fn}\, y \Rightarrow f$ of type $\mathsf{unit} \to \mathsf{unit}$ then the production



of the value $()$ has as a side effect the generation of a new computation thread for evaluating $(\mathsf{fn}\, y \Rightarrow f)\,()$.

When giving an operational semantics to a language with side-effects there are two standard approaches to retaining the information necessary to interpret them. The first, used for example in [5, 30], is to define a notion of *state* or *configuration*; these contain the program being evaluated together with auxiliary state information, and the judgements of the operational semantics apply to these configurations. The second, more common in work on process algebras, [4, 20], extends the syntax of the language being interpreted to encompass configurations. We choose the latter approach and one extra construct we add to the language is an asymmetric parallel operator, $e \parallel f$; intuitively this corresponds to the use of multi-sets in the reduction semantics of [5, 30]. As an example of the use of this extra construct the side-effect generated by the evaluation of $\mathsf{spawn}$ is reflected in our semantics by the inference:

$$
\mathsf{spawn}(\mathsf{fn}\, y \Rightarrow e) \xrightarrow{\tau} (\mathsf{fn}\, y \Rightarrow e)\,() \parallel ();
$$

one step in the evaluation of $\mathsf{spawn}(\mathsf{fn}\, y \Rightarrow e)$ leads to two expressions running in parallel, one being the spawned expression $(\mathsf{fn}\, y \Rightarrow e)\,()$ and the other the default value which results from every application of $\mathsf{spawn}$. More generally the evaluation of $\mathsf{spawn}\, e$ proceeds by the evaluation of the expression $e$ until this produces a value and then an application of an inference such as the one above. This is represented by the rule:

$$
\frac{e \xrightarrow{\surd v} e'}{\mathsf{spawn}\, e \xrightarrow{\tau} e' \parallel v\,() \parallel ()}
$$

where the well-typedness of the operational semantics will ensure that $v$ is a function of the appropriate type, $\mathsf{unit} \to \mathsf{unit}$.

With this method of representing newly created computation threads more of the rules corresponding to β-reduction in a call-by-value λ-calculus may now be given. To evaluate an application expression $e f$ first $e$ is evaluated to a value of functional form and then the evaluation of $f$ is initiated. This is represented by the the rules:

$$
\frac{e \xrightarrow{\alpha} e'}{e\, f \xrightarrow{\alpha} e'\, f} \qquad \frac{e \xrightarrow{\surd(\mathsf{fn}\, y \Rightarrow g)} e'}{e\, f \xrightarrow{\tau} e' \parallel \mathsf{let}\, y = f\, \mathsf{in}\, g}
$$

In fact we use a slightly more complicated version of the latter rule as functions are allowed to be recursive. Continuing with the evaluation of $e f$, having evaluated $e$ to the functional form $\mathsf{fn}\, y \Rightarrow g$, $f$ is evaluated to a value which is then substituted into $g$ for $y$. This is represented by the two rules:

$$
\frac{f \xrightarrow{\tau} f'}{\mathsf{let}\, x = f\, \mathsf{in}\, g \xrightarrow{\tau} \mathsf{let}\, x = f'\, \mathsf{in}\, g} \qquad \frac{f \xrightarrow{\surd v} f'}{\mathsf{let}\, x = f\, \mathsf{in}\, g \xrightarrow{\tau} f' \parallel g[v/x]}
$$

$$\frac{e \xrightarrow{\alpha} e'}{c\,e \xrightarrow{\alpha} c\,e'} \qquad \frac{e \xrightarrow{\alpha} e'}{e\,f \xrightarrow{\alpha} e'\,f} \qquad \frac{e \xrightarrow{\alpha} e'}{(e,f) \xrightarrow{\alpha} (e',f)}$$

$$\frac{e \xrightarrow{\alpha} e'}{\mathsf{if}\,e\,\mathsf{then}\,f\,\mathsf{else}\,g \xrightarrow{\alpha} \mathsf{if}\,e'\,\mathsf{then}\,f\,\mathsf{else}\,g} \qquad \frac{e \xrightarrow{\alpha} e'}{\mathsf{let}\,x = e\,\mathsf{in}\,f \xrightarrow{\alpha} \mathsf{let}\,x = e'\,\mathsf{in}\,f}$$

$$\frac{e \xrightarrow{\alpha} e'}{e \parallel f \xrightarrow{\alpha} e' \parallel f} \qquad \frac{f \xrightarrow{\alpha} f'}{e \parallel f \xrightarrow{\alpha} e \parallel f'} \qquad \frac{f \xrightarrow{\surd v} f'}{e \parallel f \xrightarrow{\surd v} e \parallel f'}$$

FIGURE 2A. Operational semantics: static rules

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

FIGURE 2B. Operational semantics: dynamic rules

$$\frac{e \xrightarrow{\surd v} e'}{c\,e \xrightarrow{\tau} e' \parallel \delta(c,v)} \qquad \frac{e \xrightarrow{\surd \mathsf{true}} e'}{\mathsf{if}\,e\,\mathsf{then}\,f\,\mathsf{else}\,g \xrightarrow{\tau} e' \parallel f} \qquad \frac{e \xrightarrow{\surd \mathsf{false}} e'}{\mathsf{if}\,e\,\mathsf{then}\,f\,\mathsf{else}\,g \xrightarrow{\tau} e' \parallel g}$$

$$\frac{e \xrightarrow{\surd v} e'}{(e,f) \xrightarrow{\tau} e' \parallel \mathsf{let}\,x = f\,\mathsf{in}\,\langle v,x\rangle} \qquad \frac{e \xrightarrow{\surd v} e'}{e\,f \xrightarrow{\tau} \mathsf{let}\,y = f\,\mathsf{in}\,g[v/x]}[v = \mathsf{fix}(x = \mathsf{fn}\,y \Rightarrow g)]$$

$$\frac{e \xrightarrow{\surd v} e'}{\mathsf{let}\,x = e\,\mathsf{in}\,f \xrightarrow{\tau} e' \parallel f[v/x]} \qquad \frac{e \xrightarrow{k!_A v} e' \quad f \xrightarrow{k?_A x} f'}{e \parallel f \xrightarrow{\tau} e' \parallel f'[v/x]} \qquad \frac{e \xrightarrow{k?_A x} e' \quad f \xrightarrow{k!_A v} f'}{e \parallel f \xrightarrow{\tau} e'[v/x] \parallel f'}$$

FIGURE 2C. Operational semantics: silent rules

$$\frac{}{v \xrightarrow{\surd v} \Lambda} \qquad \frac{}{k!_A v \xrightarrow{k!_A v} ()} \qquad \frac{}{k?_A \xrightarrow{k?_A x} x} \qquad \frac{}{\mathbf{A}v \xrightarrow{\tau} v}$$

FIGURE 2D. Operational semantics: axioms

$$\begin{aligned} \delta(\mathsf{fst}, \langle v,w\rangle) &= v & \delta(\mathsf{snd}, \langle v,w\rangle) &= w \\ \delta(\mathsf{add}, \langle m,n\rangle) &= m+n & \delta(\mathsf{mul}, \langle m,n\rangle) &= m \times n \\ \delta(\mathsf{leq}, \langle m,n\rangle) &= m \le n & & \end{aligned}$$

$$\begin{aligned} \delta(\mathsf{transmit}_A, \langle k,v\rangle) &= [k!_A v] & \delta(\mathsf{receive}_A, k) &= [k?_A] \\ \delta(\mathsf{choose}, \langle [ge_1],[ge_2]\rangle) &= [ge_1 \oplus ge_2] & \delta(\mathsf{wrap}, \langle [ge],v\rangle) &= [ge \Rightarrow v] \\ \delta(\mathsf{never}, ()) &= [\Lambda] & \delta(\mathsf{always}, v) &= [\mathbf{A}v] \\ \delta(\mathsf{spawn}, v) &= v() \parallel () & \delta(\mathsf{sync}, [ge]) &= ge \end{aligned}$$

FIGURE 2E. Operational semantics:reduction of constants



The evaluation of the application expression $cf$ is similar; $f$ is evaluated to a value and then the constant $c$ is applied to the resulting value. This is represented by the two rules

$$\frac{f \xrightarrow{\tau} f'}{cf \xrightarrow{\tau} cf'} \qquad \frac{f \xrightarrow{\surd v} f'}{cf \xrightarrow{\tau} f' \parallel \delta(c,v)}$$

Here, borrowing the notation of [30], we use the function $\delta$ to represent the effect of applying the constant $c$ to the value $v$. This effect depends on the constant in question and we have already seen one instance of this rule, for the constant $\mathsf{spawn}$, which result from the fact that $\delta(\mathsf{spawn}, v) = v() \parallel ()$. The definition of $\delta$ for all constants in the language is given in Figure 2e. For the constants associated with the base types this is self-explanatory; the others will be explained below as the constant in question is considered. Note that because of the introduction of $\parallel$ into the language we can treat all constants uniformly, unlike [30] where $\mathsf{spawn}$ and $\mathsf{sync}$ have to considered in a special manner.

In order to implement the standard left-to-right evaluation of pairs of expressions we introduce a new value $\langle v,w\rangle$ representing a pair which has been fully evaluated. Then to evaluate $(e,f)$:

- first allow $e$ to evaluate:

$$\frac{e \xrightarrow{\alpha} e'}{(e,f) \xrightarrow{\alpha} (e',f)}$$

- then when it terminates, start the evaluation of $f$:

$$\frac{e \xrightarrow{\surd v} e'}{(e,f) \xrightarrow{\tau} e' \parallel \mathsf{let}\,x = f\,\mathsf{in}\,\langle v,x\rangle}$$

These value pairs may then be used for example by being applied to functions of type $A * B$. For example the following inferences result from the definition of the function $\delta$ for the constants $\mathsf{fst}$ and $\mathsf{mul}$:

$$\frac{e \xrightarrow{\surd \langle v,w\rangle} e'}{\mathsf{fst}\,e \xrightarrow{\tau} e' \parallel v} \qquad \frac{e \xrightarrow{\surd \langle m,n\rangle} e'}{\mathsf{mul}\,e \xrightarrow{\tau} e' \parallel m \times n}.$$

It remains to explain how *delayed computations*, i.e. programs of type $A\,\mathsf{event}$, are handled. It is important to realise that expressions of type $A\,\mathsf{event}$ represent *potential* rather than actual computations and this potential can only be activated by an application of the constant $\mathsf{sync}$, of type $A\,\mathsf{event} \to A$. Thus for example the expression $\mathsf{receive}_A\,k$ is of type $A\,\mathsf{event}$ and represents a delayed computation which has the potential to receive a value of type $A$ along the channel $k$. The expression $\mathsf{sync}(\mathsf{receive}_A\,k)$ can actually receive such a value $v$ along channel $k$, or more accurately can evaluate to such a value, provided some other computation thread can send the value along channel $k$.

The semantics of $\mathsf{sync}$ is handled by introducing a new constructor for values.

For certain kinds of expressions $ge$ of type A, which we call *guarded expressions*, let $[ge]$ be a value of type $A\,\mathsf{event}$; this represents a *delayed computation* which when launched initiates a new computation thread which evaluates the expression $ge$. Then the expression $\mathsf{sync}\,[ge]$ reduces in one step to the expression $ge$. More generally the evaluation of the expression $\mathsf{sync}\,e$ proceeds as follows:

- First evaluate $e$ until it can produce a value:

$$\frac{e \xrightarrow{\tau} e'}{\mathsf{sync}\,e \xrightarrow{\tau} \mathsf{sync}\,e'}$$

- then launch the resulting delayed computation:

$$\frac{e \xrightarrow{\surd[ge]} e'}{\mathsf{sync}\,e \xrightarrow{\tau} e' \parallel ge}$$

Note that here, as always, the production of a value may have as a side-effect the generation of a new computation thread $e'$ and this is launched concurrently with the delayed computation $ge$. Also both of these rules are instances of more general rules already considered. The first is obtained from the rule for the evaluation of applications of the form $ce$ and the second by defining $\delta(\mathsf{sync},[ge])$ to be $ge$.

The precise syntax for guarded expressions will emerge by considering what types of values of the form $[e]$ can result from the evaluation of expressions of type $\mathsf{event}$ from the basic language $\mu$CML. The constant $\mathsf{receive}_A$ is of type $\mathsf{chan} \to A\,\mathsf{event}$ and therefore the evaluation of the expression $\mathsf{receive}_A\,e$ proceeds by first evaluating $e$ to a value of type $\mathsf{chan}$ until it returns a value $k$, and then returning a delayed computation consisting of an event which can receive any value of type $A$ on the channel $k$. To represent this event we extend the syntax further by letting $k?_A$ be a guarded expression for any $k$ and $A$, with the associated rule:

$$\frac{e \xrightarrow{\surd k} e'}{\mathsf{receive}_A\,e \xrightarrow{\tau} e' \parallel [k?_A]}$$

The construct $\mathsf{transmit}_A$ is handled in a similar manner, using guarded expressions of the form $k!_A v$:

$$\frac{e \xrightarrow{\surd\langle k,v\rangle} e'}{\mathsf{transmit}_A\,e \xrightarrow{\tau} e' \parallel [k!_A v]}$$

It is these two new expressions $k?_A$ and $k!_A v$ which perform communication between computation threads. Formally $k!_A v$ is of type $\mathsf{unit}$ and we have the axiom:

$$\frac{}{k!_A v \xrightarrow{k!_A v} ()}$$

Intuitively this may be read as $k!_A v$ evaluates in one step to the expression $()$ and this evaluation has as a side effect the transmission of the value $v$ to the channel $k$. The input rule is slightly more complicated. Because these communication moves are propagated through various contexts it is technically convenient to have the inference rule:



$$\frac{}{k?_A \xrightarrow{k?_A x} x}$$

Therefore in general input moves are of the form $e \xrightarrow{k?_A x} f$ where $\vdash e : B$ and $x : A \vdash f : B$. Communication can now be modelled as in CCS and CSP by the simultaneous occurrence of input and output actions:

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

There remain four constructs for *delayed computations* to be explained. The first, $\mathsf{never}$ of type $\mathsf{unit} \to A\,\mathsf{event}$, is handled by the introduction of the guarded expression $\Lambda$, representing a deadlocked evaluation, together with the inference rule:

$$\frac{e \xrightarrow{\surd()} e'}{\mathsf{never}\,e \xrightarrow{\tau} e' \parallel [\Lambda]}$$

obtained, once more, by defining $\delta(\mathsf{never},())$ to be $[\Lambda]$.

The constant $\mathsf{wrap}$ is of type $A\,\mathsf{event} * (A \to B) \to B\,\mathsf{event}$. The evaluation of $\mathsf{wrap}\,e$ proceeds in the standard way by evaluating $e$ until it produces a value, which must be of the form $\langle [ge], v\rangle$, where $ge$ is a guarded expression of type $A$ and $v$ has type $A \to B$. Then the evaluation of $\mathsf{wrap}\,e$ continues by the construction of the new *delayed computation* $[ge \Rightarrow v]$. Bearing in mind the fact that the production of values can generate new computation threads, this is formally represented by the inference rule:

$$\frac{e \xrightarrow{\surd\langle [ge],v\rangle} e'}{\mathsf{wrap}\,e \xrightarrow{\tau} e' \parallel [ge \Rightarrow v]}$$

The guarded expression $ge \Rightarrow v$ is a *wrapper* which applies $v$ to the result of evaluating $ge$:

$$\frac{ge \xrightarrow{\alpha} e}{ge \Rightarrow v \xrightarrow{\alpha} v\,e}$$

The $\mathsf{always}$ construct, of type $A \to A\,\mathsf{event}$, evaluates its argument to a value $v$, and then returns trivial delayed computation; this computation, when activated, immediately evaluates to the value $v$. In order to represent these trivial computations we introduce a new constructor for guarded expressions, $\mathbf{A}$ and the semantics of $\mathsf{always}$ is then captured by the rule:

$$\frac{e \xrightarrow{\surd v} e'}{\mathsf{always}\,e \xrightarrow{\tau} e' \parallel [\mathbf{A} v]}$$

Since $\mathbf{A}v$ immediately evaluates to the constant $v$ we have:

$$\frac{}{\mathbf{A}v \xrightarrow{\tau} v}$$

The choice construct $\mathsf{choose}\,e$ is a choice between *delayed computations* as $\mathsf{choose}$ has the type $A\,\mathsf{event} * A\,\mathsf{event} \to A\,\mathsf{event}$. To interpret it we introduce a new choice constructor $ge_1 \oplus ge_2$ where $ge_1$ and $ge_2$ are guarded expressions of the same type. Then $\mathsf{choose}\,e$ proceeds by evaluating $e$ until it can produce a value, which must be of the form $\langle [ge_1], [ge_2] \rangle$, and the evaluation continues by constructing the *delayed computation* $[ge_1 \oplus ge_2]$. This is represented by the rule:

$$\frac{e \xrightarrow{\surd\langle [ge_1],[ge_2]\rangle} e'}{\mathsf{choose}\,e \xrightarrow{\tau} e' \parallel [ge_1 \oplus ge_2]}$$

The notation $\oplus$, introduced in [30], is unfortunate, as it is used in [14] to represent the *internal choice* between processes whereas here it represents *external choice:* we have the following auxiliary rules , which are the same as CCS summation:

$$\frac{ge_1 \xrightarrow{\alpha} e}{ge_1 \oplus ge_2 \xrightarrow{\alpha} e} \qquad \frac{ge_2 \xrightarrow{\alpha} e}{ge_1 \oplus ge_2 \xrightarrow{\alpha} e}$$

This ends our informal description of the operational semantics of $\mu$CML. We now summarise, giving the precise definitions of the new syntax. For the purposes of comparison with the reduction semantics of $\lambda_{cv}$, [30], it is convenient to view the extension to $\mu$CML in two stages. The first is obtained by adding the new syntactic category of guarded expressions, and two new constructors for values:

$$v \in Val ::= \cdots \mid \langle v, v \rangle \mid [ge]$$
$$ge \in GExp ::= v!_A v \mid v? \mid ge \Rightarrow v \mid ge \oplus ge \mid \Lambda \mid \mathbf{A}v$$

The resulting language we call $\mu\mathrm{CML}^{cv}$, as it corresponds very closely to Reppy's $\lambda_{cv}$. A precise comparison is given in Section 6. The final language, $\mu\mathrm{CML}^{+}$, is obtained by extending $\mu\mathrm{CML}^{cv}$ with:

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

and type judgements for all the extra constructs is given in Figure 3.

The operational semantics is given as a set of transition relations over closed expressions from $\mu\mathrm{CML}^{+}$. These transition relations have as labels *Label*:

$$a ::= v!_A v \mid v?_A x \qquad \alpha ::= a \mid \tau \qquad l ::= \alpha \mid \surd v$$

which are typed with judgements $\vdash l : A$ in Figure 4, and are defined to be the least relations satisfying the rules in Figure 2. The rules are divided into three parts. The first gives the set of context rules, showing when moves may be propagated through certain contexts; the second give the reduction rules while the third contains the axioms.



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

$$\frac{\Gamma \vdash v : \mathsf{chan} \quad \Gamma \vdash w : A}{\Gamma \vdash v!_A w : \mathsf{unit}} \qquad \frac{\Gamma \vdash v : \mathsf{chan}}{\Gamma \vdash v?_A : A} \qquad \frac{\Gamma \vdash ge : A \quad \Gamma \vdash v : A \to B}{\Gamma \vdash ge \Rightarrow v : B}$$

$$\frac{\Gamma \vdash ge_1 : A \quad \Gamma \vdash ge_2 : A}{\Gamma \vdash ge_1 \oplus ge_2 : A} \qquad \frac{}{\Gamma \vdash \Lambda : A} \qquad \frac{\Gamma \vdash v : A}{\Gamma \vdash \mathbf{A}v : A}$$

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

FIGURE 3. Type rules for extra $\mu\mathrm{CML}^{+}$ constructs

$$\frac{}{\Gamma \vdash \tau : A} \qquad \frac{\Gamma \vdash v : \mathsf{chan} \quad \Gamma \vdash w : B}{\Gamma \vdash v!_B w : A} \qquad \frac{\Gamma \vdash v : \mathsf{chan}}{\Gamma \vdash v?_B x : A} \qquad \frac{\Gamma \vdash v : A}{\Gamma \vdash \surd v : A}$$

FIGURE 4. Type rules for labels

It is worth pointing out that the context rules are asymmetric for the propagation of value production though the context $\parallel$; in $e \parallel f$ only the computation thread $f$ can produce a value. This is in agreement with the reduction semantics of [30] where in a given state represented by a multi-set of expressions only one distinguished expression is allowed to produce a value. Also in the rule for application, the evaluation of $e\,f$ is somewhat more complicated than previously stated; values of functional type all involve the fix point operator and these fix points are automatically unfolded at the point of application.

We end this section with a Subject Reduction Theorem for our semantics:

THEOREM 2.1. *For every closed expression e in $\mu\mathrm{CML}^{+}$*

- *if* $e \xrightarrow{l} e'$ *and* $\vdash e : A$ *then* $\vdash l : A$,
- *if* $e \xrightarrow{\tau} e'$ *and* $\vdash e : A$ *then* $\vdash e' : A$,
- *if* $e \xrightarrow{\surd v} e'$ *and* $\vdash e : A$ *then* $\vdash e' : A$,
- *if* $e \xrightarrow{k?_B x} e'$ *and* $\vdash e : A$ *then* $x : B \vdash e' : A$, *and*
- *if* $e \xrightarrow{k!_B v} e'$ *and* $\vdash e : A$ *then* $\vdash e' : A$.

PROOF. By rule induction on the inferences. □

## 3 Weak Bisimulation Equivalence

In this section we demonstrate the usefulness of our operational semantics by providing $\mu\mathrm{CML}^{+}$ with an appropriate version of bisimulation equivalence. We discuss a range of possible bisimulation based equivalences and eventually propose a new variation called *hereditary bisimulation equivalence*, which we feel is most suited to $\mu$CML$^+$.

We first show how to adapt the notion of strong bisimulation equivalence to $\mu$CML$^+$. Since our language is typed it is more convenient to define the equivalence in terms of *type-indexed* families of relations. Moreover since the operational semantics uses actions of the form $e \xrightarrow{k?_B x} f$ where $f$ may be an open expression we need to consider relations over open expressions. Let an *open type-indexed* relation $\mathcal{R}$ be a family of relations $\mathcal{R}_{\Gamma,A}$ such that if $e \mathrel{\mathcal{R}_{\Gamma,A}} f$ then $\Gamma \vdash e : A$ and $\Gamma \vdash f : A$. We will often elide the subscripts from relations, for example writing $e \mathrel{\mathcal{R}} f$ for $e \mathrel{\mathcal{R}_{\Gamma,A}} f$ when context makes the type 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:

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

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

- if $v \mathrel{\mathcal{R}_A} w$ and $A$ is a base type then $v = w$,
- if $\langle v_1, v_2 \rangle \mathrel{\mathcal{R}_{A_1 * A_2}} \langle w_1, w_2 \rangle$ then $v_i \mathrel{\mathcal{R}_{A_i}} w_i$,
- if $[ge_1] \mathrel{\mathcal{R}_{A\,\mathsf{event}}} [ge_2]$ then $ge_1 \mathrel{\mathcal{R}_A} ge_2$, and
- if $v \mathrel{\mathcal{R}_{A \to B}} v'$ then for all $\vdash w : A$ we have $v\,w \mathrel{\mathcal{R}_B} v'\,w$.

With this notation we can now define strong bisimulations over $\mu$CML$^+$ expressions. 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:

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

Note the use of the open extension $\mathcal{R}^\circ$. This means, for example, that if $e_1 \mathrel{\mathcal{R}} e_2$ we require that the move $e_1 \xrightarrow{k?_B x} f_1$ be matched by a move $e_2 \xrightarrow{k?_B x} f_2$ where $f_2$ is such that for all values $v$ of the appropriate type $f_1[v/x] \mathrel{\mathcal{R}} f_2[v/x]$. Thus in the terminology of [22] 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 3.1. *$\sim^1$ 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. $\square$



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

$$\mathsf{add}(1,2) \sim^1 \mathsf{add}(2,1)$$

however, sending the thunked expressions on channel $k$ we get:

$$k!(\mathsf{fn}\, x \Rightarrow \mathsf{add}(1,2)) \not\sim^1 k!(\mathsf{fn}\, x \Rightarrow \mathsf{add}(2,1))$$

since the definition of strong bisimulation demands that the actions performed by expressions match up to syntactic identity. This counter-example can also be reproduced using only $\mu$CML contexts:

$$\mathsf{sync}(\mathsf{transmit}(k, \mathsf{fn}\, x \Rightarrow \mathsf{add}(1,2))) \not\sim^1 \mathsf{sync}(\mathsf{transmit}(k, \mathsf{fn}\, x \Rightarrow \mathsf{add}(2,1)))$$

since the lhs can perform the move:

$$\mathsf{sync}(\mathsf{transmit}(k, \mathsf{fn}\, x \Rightarrow \mathsf{add}(1,2))) \xrightarrow{k!(\mathsf{fn}\, x \Rightarrow \mathsf{add}(1,2))} ()$$

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

$$\mathsf{sync}(\mathsf{transmit}(k, \mathsf{fn}\, x \Rightarrow \mathsf{add}(2,1))) \xrightarrow{k!(\mathsf{fn}\, x \Rightarrow \mathsf{add}(2,1))} ()$$

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:

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

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

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

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

PROPOSITION 3.2. *$\sim^h$ is a congruence.*

PROOF. Use a similar technique to the proof of Proposition 3.1 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. $\square$

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 of β-reduction such as $Id\,0 = 0$ where $Id$ denotes the identity function ($\mathsf{fn}\ x \Rightarrow x$). We require the looser *weak bisimulation* which allows τ-actions to be ignored.

This in turn requires some more notation. Let $\stackrel{\varepsilon}{\Longrightarrow}$ be the reflexive transitive closure of $\xrightarrow{\tau}$, and let $\stackrel{l}{\Longrightarrow}$ be $\stackrel{\varepsilon}{\Longrightarrow}\xrightarrow{l}$ (i.e. any sequence of silent action followed by an $l$ action). Note that we are *not* allowing silent actions after the $l$ action. Let $\stackrel{\hat{l}}{\Longrightarrow}$ be $\stackrel{\varepsilon}{\Longrightarrow}$ if $l = \tau$ and $\stackrel{l}{\Longrightarrow}$ otherwise. Then $\mathcal{R}$ is a *first-order weak simulation* iff it is structure-preserving and the following diagram can be completed:

$$\begin{array}{ccc} e_1 & \mathcal{R} & e_2 \\ {\scriptstyle l}\downarrow & & \\ e_1' & & \end{array} \qquad \text{as} \qquad \begin{array}{ccc} e_1 & \mathcal{R} & e_2 \\ {\scriptstyle l}\downarrow & & \Downarrow{\scriptstyle \hat{l}} \\ e_1' & \mathcal{R}^\circ & e_2' \end{array}$$

Let $\approx^1$ be the largest first-order strong bisimulation.

PROPOSITION 3.3. *$\approx^1$ is an equivalence.*

PROOF. Similar to the proof of Proposition 3.1. □

Unfortunately, $\approx^1$ is not a congruence, for the same reason as $\sim^1$, and so we can attempt the same fix. $\mathcal{R}$ is a *higher-order weak simulation* iff it is structure-preserving and the following diagram can be completed:

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

Let $\approx^h$ be the largest higher-order weak bisimulation.

PROPOSITION 3.4. *$\approx^h$ is an equivalence.*

PROOF. Similar to the proof of Proposition 3.1. □

However, $\approx^h$ is not a congruence, for the usual reason that weak bisimulation equivalence $\approx$ is not a congruence for CCS summation. Recall from [20] that $Nil \approx \tau.Nil$ but $k!0 + Nil \not\approx k!0 + \tau.Nil$. We can duplicate this counter-example in $\mu$CML$^+$ since the CCS operator $+$ corresponds to the $\mu$CML$^+$ operator $\oplus$ and $Nil$ corresponds to $\Lambda$. However $\oplus$ may only be applied to *guarded expressions* and therefore we need a *guarded expression* which behaves like $\tau.Nil$; the required



expression is $\mathbf{A}[\Lambda] \Rightarrow \mathsf{sync}$. Thus:

$$\Lambda \approx^h \mathbf{A}[\Lambda] \Rightarrow \mathsf{sync}$$

since the rhs has only one reduction:

$$\begin{aligned} &\mathbf{A}[\Lambda] \Rightarrow \mathsf{sync} \\ &\quad \xrightarrow{\tau} \mathsf{sync}[\Lambda] \\ &\quad \xrightarrow{\tau} \Lambda \end{aligned}$$

but:

$$\Lambda \oplus k!0 \not\approx^h (\mathbf{A}[\Lambda] \Rightarrow \mathsf{sync}) \oplus k!0$$

because:

$$\begin{aligned} &(\mathbf{A}[\Lambda] \Rightarrow \mathsf{sync}) \oplus k!0 \\ &\quad \xrightarrow{\tau} \mathsf{sync}[\Lambda] \\ &\quad \xrightarrow{\tau} \Lambda \end{aligned}$$

This counter-example can also be replicated using the restricted syntax of $\mu$CML. We have:

$$\mathsf{never}() \approx^h \mathsf{wrap}(\mathsf{always}(\mathsf{never}()), \mathsf{sync})$$

since the lhs has only one reduction:

$$\mathsf{never}() \stackrel{\surd[\Lambda]}{\Longrightarrow} \Lambda$$

and the rhs can match this with:

$$\mathsf{wrap}(\mathsf{always}(\mathsf{never}()), \mathsf{sync}) \xRightarrow{\surd[\mathbf{A}[\Lambda] \Rightarrow \mathsf{sync}]} \Lambda$$

and we have seen:

$$\Lambda \approx^h \mathbf{A}[\Lambda] \Rightarrow \mathsf{sync}\,.$$

However:

$$\begin{aligned} &\mathsf{sync}(\mathsf{choose}(\mathsf{never}(), \mathsf{transmit}(k, 0))) \\ &\quad \not\approx^h \mathsf{sync}(\mathsf{choose}(\mathsf{wrap}(\mathsf{always}(\mathsf{never}()), \mathsf{sync}), \mathsf{transmit}(k, 0))) \end{aligned}$$

since the lhs has only one reduction:

$$\begin{aligned} &\mathsf{sync}(\mathsf{choose}(\mathsf{never}(), \mathsf{transmit}(k, 0))) \\ &\quad \stackrel{\tau}{\Longrightarrow} \Lambda \oplus k!0 \end{aligned}$$

whereas the rhs has the reduction:

$$\begin{aligned} &\mathsf{sync}(\mathsf{choose}(\mathsf{wrap}(\mathsf{always}(\mathsf{never}()), \mathsf{sync}), \mathsf{transmit}(k, 0))) \\ &\quad \stackrel{\tau}{\Longrightarrow} (\mathbf{A}[\Lambda] \Rightarrow \mathsf{sync}) \oplus k!0 \end{aligned}$$

A first attempt to rectify this is to adapt Milner's observational equivalence for $\mu$CML, and to define $=^h$ as the smallest symmetric relation such that the following diagram can be completed:

$$\begin{tikzcd} e_1 \arrow[d, "l_1"'] & =^h & e_2 \\ e_1' & & \end{tikzcd} \quad \text{as} \quad \begin{tikzcd} e_1 \arrow[d, "l_1"'] & =^h & e_2 \arrow[d, Rightarrow, "l_2"'] \\ e_1' & \approx^h & e_2' \end{tikzcd} \text{ where } l_1 \approx^{hl} l_2$$

PROPOSITION 3.5. *$=^h$ is an equivalence.*

PROOF. Similar to the proof of Proposition 3.1. □

This attempt fails, however, since it only looks at the first move of a process, and not at the first moves of any processes in its transitions. Thus, the above $\mu$CML counter-example for $\approx^h$ being a congruence also applies to $=^h$. This failure was first noted by Thomsen [32] for CHOCS.

Thomsen's solution to this problem is to require that $\tau$-moves can always be matched by at least one $\tau$-move, which produces his definition of an *irreflexive simulation* as a structure-preserving relation where the following diagram can be completed:

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

Let $\approx^i$ be the largest irreflexive bisimulation.

PROPOSITION 3.6. *$\approx^i$ is a congruence.*

PROOF. The proof that $\approx^i$ is an equivalence is similar to the proof of Proposition 3.1. The proof that it is a congruence is similar to the proof of Theorem 4.7 in the next section. □

However this relation is rather too strong for many purposes, for example $\mathsf{add}(1,2) \not\approx^i \mathsf{add}(1,\mathsf{add}(1,1))$ since the rhs can perform more $\tau$-moves than the lhs. This is similar to the problem in CHOCS where $a.\tau.P \not\approx^i a.P$.

In order to find an appropriate definition of bisimulation for $\mu$CML, we observe that $\mu$CML only allows $\oplus$ to be used on *guarded expressions*, and not on arbitrary expressions. We can thus ignore the initial $\tau$-moves of all expressions *except* for guarded expressions. For this reason, we have to provide *two* equivalences: one on terms where we are not interested in initial $\tau$-moves, and one on terms where we are.



A pair of closed type-indexed relations $\mathcal{R} = (\mathcal{R}^n, \mathcal{R}^s)$ form a *hereditary simulation* (we call $\mathcal{R}^n$ an *insensitive simulation* and $\mathcal{R}^s$ a *sensitive simulation*) iff $\mathcal{R}^s$ is structure-preserving and we can complete the following diagrams:

$$\begin{tikzcd} e_1 \arrow[d, "l_1"'] & \mathcal{R}^n & e_2 \\ e_1' & & \end{tikzcd} \quad \text{as} \quad \begin{tikzcd} e_1 \arrow[d, "l_1"'] & \mathcal{R}^n & e_2 \arrow[d, Rightarrow, "\hat{l}_2"'] \\ e_1' & \mathcal{R}^{n\circ} & e_2' \end{tikzcd} \text{ where } l_1 \mathrel{\mathcal{R}^{sl}} l_2$$

and:

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

Let $(\approx^n, \approx^s)$ be the largest hereditary bisimulation.

In the operational semantics of expressions from $\mu$CML guarded expressions are introduced as components to *Labels* and never as residuals. This explains why in the definition of $\approx^n$ labels are compared with respect to the sensitive relation $\approx^s$ whereas the the insensitive relation is used for the residuals. For example, if $ge_1 \approx^n ge_2$ then we have:

$$(\mathsf{fn}\, x \Rightarrow ge_1) \approx^n (\mathsf{fn}\, x \Rightarrow ge_2)$$

since once either side is applied to an argument, their first action will be a $\tau$-step. On the other hand:

$$[ge_1] \not\approx^n [ge_2]$$

THEOREM 3.7. *$\approx^s$ is a congruence for $\mu$CML$^+$, and $\approx^n$ is a congruence for $\mu$CML.*

PROOF. The proof that $\approx^s$ and $\approx^n$ are equivalences is similar to the proof of Proposition 3.1. The proof that they form congruences is the subject of the next section. □

PROPOSITION 3.8. *The equivalences on $\mu$CML$^+$ have the following strict in-*

*clusions:*

$$\begin{tikzcd}
\sim^1 \arrow[rrr, hook] \arrow[d, hook] & & & \approx^1 \arrow[d, hook] \\
\sim^h \arrow[r, hook] & \approx^i \arrow[r, hook] & \approx^s \arrow[r, hook] \arrow[d, hook] & \approx^n \arrow[d, hook] \\
& & =^h \arrow[r, hook] & \approx^h
\end{tikzcd}$$

PROOF. For each inclusion, show that the first bisimulation satisfies the condition required to be the second form of bisimulation. To show that the inclusions are strict, we use the following examples:

$$\begin{aligned}
(\mathsf{fn}\,x \Rightarrow \mathsf{add}(1,2)) &\sim^h\not\sim^1 (\mathsf{fn}\,x \Rightarrow \mathsf{add}(2,1)) \\
1 &\approx^1\not\sim^1 \mathsf{let}\,x = 1\,\mathsf{in}\,x \\
\mathsf{choose}(\mathsf{receive}\,k, \mathsf{tau}(\mathsf{receive}\,k)) &\approx^i\not\sim^h \mathsf{tau}(\mathsf{receive}\,k) \\
\mathsf{add}(1,2) &\approx^s\not\approx^i \mathsf{add}(1,\mathsf{add}(1,1)) \\
1 &\approx^n\not\approx^s \mathsf{let}\,x = 1\,\mathsf{in}\,x \\
\mathsf{never}() &\approx^h\not\approx^n \mathsf{tau}(\mathsf{never}()) \\
1 &\approx^h\neq^h \mathsf{let}\,x = 1\,\mathsf{in}\,x
\end{aligned}$$

where:

$$\mathsf{tau} = \mathsf{fn}\,x \Rightarrow \mathsf{wrap}(\mathsf{always}\,x, \mathsf{sync})$$

(Note that this settles an open question [32] of Thomsen's as to whether $\approx^i$ is the largest congruence contained in $\approx^h$.) □

It is the operator $\oplus$ which differentiates between the two equivalences $\approx^n$ and $\approx^h$. However in order to demonstrate the difference we need to be able to apply $\oplus$ to guarded expressions which can spontaneously evolve, i.e. perform $\tau$-moves. The only $\mu$CML$^+$ constructor for guarded expressions which allows this is $\mathbf{A}$, and in turn occurrences of this can only be generated by the $\mu$CML constructor $\mathsf{always}$. Therefore:

PROPOSITION 3.9. *For the subset of $\mu$CML$^+$ without* $\mathsf{always}$ *and* $\mathbf{A}$*, $\approx^n$ is the same as $\approx^h$, and $\approx^s$ is the same as $=^h$.*

PROOF. From Proposition 3.8 $\approx^n \subseteq \approx^h$.



For the subset of $\mu$CML$^+$ without $\mathsf{always}$ and $\mathbf{A}$, define $\mathcal{R}$ as:

$$\{(v,w) \mid v \approx^h w\} \cup \{(ge_1, ge_2) \mid ge_1 \approx^h ge_2\} \cup \{(v_1\,w, v_2\,w) \mid v_1 \approx^h v_2\}$$

Then since no event without $\mathbf{A}$ can perform a $\tau$-move, and since the only initial moves of $v_i\,w$ are $\beta$-reductions, we can show that $(\approx^h, \mathcal{R})$ form an hereditary bisimulation, and so $\approx^h \subseteq \approx^n$. From this it is routine to show that $\approx^s = =^h$. □

Unfortunately we have not been able to show that $\approx^n$ is the largest $\mu$CML congruence contained in weak higher-order bisimulation equivalence. However we do have the following characterisation:

THEOREM 3.10. *$\approx^n$ is the largest higher-order weak bisimulation which respects $\mu$CML contexts.*

PROOF. By definition, $\approx^n$ is a higher-order weak bisimulation, and we have shown that it respects $\mu$CML contexts. All that remains is to show that it is the largest such.

Let $\mathcal{R}$ be a higher-order weak bisimulation which respects $\mu$CML contexts. Then define:

$$\begin{aligned}
\mathcal{R}^n &= \mathcal{R} \cup \{v_1\,w, e_2) \mid v_1\,\mathcal{R}\,v_2, v_2\,w \xrightarrow{\tau} e_2\} \cup \{e_1, v_2\,w) \mid v_1\,\mathcal{R}\,v_2, v_1\,w \xrightarrow{\tau} e_1\} \\
\mathcal{R}^s &= \{(v,w) \mid v\,\mathcal{R}\,w\} \cup \{(ge_1, ge_2) \mid [ge_1]\,\mathcal{R}\,[ge_2]\} \cup \{(v_1\,w, v_2\,w) \mid v_1\,\mathcal{R}\,v_2\}
\end{aligned}$$

We will now show that $(\mathcal{R}^n, \mathcal{R}^s)$ forms a hereditary simulation, from which we can deduce $\mathcal{R} \subseteq \mathcal{R}^n \subseteq \approx^n$.

First, we note that $\mathcal{R}^s$ is structure preserving, and that $\mathcal{R}^{s\,l} = \mathcal{R}^l$.

Then we show that we can complete the required diagrams for $(\mathcal{R}^n, \mathcal{R}^s)$ to be a hereditary simulation. The only tricky case is if:

$$\begin{tikzcd}
ge_1 \arrow[d, "l_1"'] & \mathcal{R}^s & ge_2 \\
e_1 & &
\end{tikzcd}$$

in which case, by the definition of $\mathcal{R}^s$, $[ge_1]\,\mathcal{R}\,[ge_2]$, and since $\mathcal{R}$ respects $\mu$CML contexts we have (for fresh $k$):

$$\begin{tikzcd}
\mathsf{choose}([ge_1], \mathsf{receive}\,k) \arrow[d, Rightarrow, "\surd[ge_1 \oplus k?]"'] & \mathcal{R} & \mathsf{choose}([ge_2], \mathsf{receive}\,k) \arrow[d, Rightarrow, "\surd[ge_2 \oplus k?]"'] \\
\Lambda & \mathcal{R} & \Lambda
\end{tikzcd}$$

and since $\mathcal{R}$ is a higher-order weak bisimulation, we have:

$$\begin{tikzcd} ge_1 \oplus k? \arrow[d, "l_1"'] & \mathcal{R} & ge_2 \oplus k? \\ e_1 & & \end{tikzcd}$$

which can be completed as:

$$\begin{tikzcd} ge_1 \oplus k? \arrow[d, "l_1"'] & \mathcal{R} & ge_2 \oplus k? \arrow[d, Rightarrow, "\hat{l}_2"'] \\ e_1 & \mathcal{R} & e_2 \end{tikzcd} \quad \text{where } l_1 \; \mathcal{R}^l \; l_2$$

but since $e_1 \stackrel{k?x}{\not\longrightarrow}$ and $l_1 \neq k?x$, we have $e_2 \stackrel{k?x}{\not\Longrightarrow}$ and $l_2 \neq k?x$, and so:

$$\begin{tikzcd} ge_1 \arrow[d, "l_1"'] & \mathcal{R}^s & ge_2 \arrow[d, Rightarrow, "l_2"'] \\ e_1 & \mathcal{R} & e_2 \end{tikzcd} \quad \text{where } l_1 \; \mathcal{R}^{sl} \; l_2$$

The other cases are simpler, and so $(\mathcal{R}^n, \mathcal{R}^s)$ is a hereditary bisimulation. Thus $\mathcal{R} \subseteq \mathcal{R}^n \subseteq \approx^n$, and so $\approx^n$ is the largest higher-order weak bisimulation which respects $\mu$CML contexts. □

## 4 Bisimulation as a congruence

To serve as the basis of a useful semantic theory of $\mu$CML, bisimulation should be preserved by all of the constructs of the language. In this section we will show that $\approx^s$ is a congruence for $\mu\text{CML}^+$, and that $\approx^n$ is a congruence for $\mu$CML.

Unfortunately, this proof is not straightforward, due to the higher-order nature of hereditary bisimulation. The problem is not unique to $\mu$CML, and it occurs in many higher-order languages, for example Gordon's [11] operational semantics for the typed λ-calculus, Howe's [17] treatment of the lazy λ-calculus, and Thomsen's [32] Calculus of Higher-Order Communicating Systems (CHOCS).

The difficulty is in finding the right form of induction to use, when all of the standard inductions (for example on structure of terms, on number of τ-moves, on structure of proof) fail. For example, the proof of congruence for CHOCS [32, Prop. 6.6] adapts Milner's technique [20, Theorem 8, p. 155] but uses a non-well-founded induction. It seems that any inductive proof that weak bisimulation is a



congruence for higher-order languages requires an induction on both syntax *and* proof structure. The usual methods of performing nested induction fail in this case, and so another method of performing simultaneous induction is required. Fortunately this is achieved by a technique developed for the lazy λ-calculus by Howe [17].

We shall apply Howe's technique to show that $\approx^s$ is a congruence for $\mu\text{CML}^+$, and that $\approx^n$ is a congruence for $\mu\text{CML}^+$ without $[ge]$ and $ge_1 \oplus ge_2$. This particular application is made complicated by the fact that we have to deal a pair of relations, $(\approx^n, \approx^s)$ which are defined in terms of each other. So although we follow the general proof method used in [17] and the notation of [11], the various technical definitions about relations which follow will apply to pairs of relations of the form $\mathcal{R} = (\mathcal{R}^n, \mathcal{R}^s)$ with $\mathcal{R}^s \subseteq \mathcal{R}^n$. We will continue to apply the usual operations associated with relations, such as composition, under the assumption that such operations are applied pointwise.

Define a *context* to be given by the grammar:

$$\begin{aligned} C ::= {} & \cdot_i \mid e \mid c\,C \mid \mathsf{if}\,C\,\mathsf{then}\,C\,\mathsf{else}\,C \mid (C,C) \mid \mathsf{let}\,x = C\,\mathsf{in}\,C \\ & \mid C\,C \mid \mathsf{fix}(x = \mathsf{fn}\;y \Rightarrow C) \mid \langle C,C \rangle \\ & \mid [C] \mid C!_A C \mid C?_A \mid C \Rightarrow C \mid C \oplus C \mid \mathbf{A}C \mid C \parallel C \end{aligned}$$

Let $C[\vec{e}]$ be the term given by replacing each 'hole' $\cdot_i$ by the term $e_i$ (unlike substitution, we allow for capture of free variables). An equivalence $\mathcal{R}$ is an *congruence* iff $e_i \; \mathcal{R} \; f_i$ implies $C[\vec{e}] \; \mathcal{R} \; C[\vec{f}]$.

Define a *uneventful context* to be one which does not use $[C]$ or $C \oplus C$, that is one given by the grammar:

$$\begin{aligned} C_n ::= {} & \cdot_i \mid e \mid c\,C_n \mid \mathsf{if}\,C_n\,\mathsf{then}\,C_n\,\mathsf{else}\,C_n \mid (C_n,C_n) \mid \mathsf{let}\,x = C_n\,\mathsf{in}\,C_n \\ & \mid C_n\,C_n \mid \mathsf{fix}(x = \mathsf{fn}\;y \Rightarrow C_n) \mid \langle C_n,C_n \rangle \\ & \mid C_n!_A C_n \mid C_n?_A \mid C_n \Rightarrow C_n \mid \mathbf{A}C_n \mid C_n \parallel C_n \end{aligned}$$

An equivalence $\mathcal{R}$ is an *uneventful congruence* iff $e_i \; \mathcal{R} \; f_i$ implies $C_n[\vec{e}] \; \mathcal{R} \; C_n[\vec{f}]$. Note that any $\mu$CML context is an uneventful context, and so any uneventful congruence is a congruence for $\mu$CML. So we concentrate on showing that $\approx^s$ is a congruence, and $\approx^n$ is an uneventful congruence.

Define the *one-level deep* contexts with the grammar:

$$\begin{aligned} D ::= {} & x \mid l \mid c\,\cdot_1 \mid \mathsf{if}\,\cdot_1\,\mathsf{then}\,\cdot_2\,\mathsf{else}\,\cdot_3 \mid (\cdot_1, \cdot_2) \mid \mathsf{let}\,x = \cdot_1\,\mathsf{in}\,\cdot_2 \\ & \mid \cdot_1\,\cdot_2 \mid \mathsf{fix}(x = \mathsf{fn}\;y \Rightarrow \cdot_1) \mid \langle \cdot_1, \cdot_2 \rangle \\ & \mid [\cdot_1] \mid \cdot_1 !_A \cdot_2 \mid \cdot_1 ?_A \mid \cdot_1 \Rightarrow \cdot_2 \mid \cdot_1 \oplus \cdot_2 \mid \mathbf{A}\cdot_1 \mid \cdot_1 \parallel \cdot_2 \end{aligned}$$

Let $D_n$ range over uneventful one-level deep contexts.

For any pair of relations $\mathcal{R} = (\mathcal{R}^n, \mathcal{R}^s)$ with $\mathcal{R}^s \subseteq \mathcal{R}^n$, let its *compatible* *refinement*, $\widehat{\mathcal{R}}$ be defined:

$$\begin{aligned}
\widehat{\mathcal{R}}^n &= \{(D_n[\vec{e}], D_n[\vec{f}]) \mid e_i \mathrel{\mathcal{R}^n} f_i\} \cup \widehat{\mathcal{R}}^s \\
\widehat{\mathcal{R}}^s &= \{(D[\vec{e}], D[\vec{f}]) \mid e_i \mathrel{\mathcal{R}^s} f_i\} \\
&\quad \cup \{(\mathsf{fix}(x = \mathsf{fn}\, y \Rightarrow e), \mathsf{fix}(x = \mathsf{fn}\, y \Rightarrow f)) \mid e \mathrel{\mathcal{R}^n} f\}
\end{aligned}$$

This definition is rather different from Howe's and Gordon's definition of $\widehat{\mathcal{R}} = \{(D[\vec{e}], D[\vec{f}]) \mid e_i \mathrel{\mathcal{R}} f_i\}$. The differences are that:

- $\approx^n$ is not a congruence, it is only an uneventful congruence, so we only close $\widehat{\mathcal{R}}^n$ under uneventful one-level deep contexts rather than arbitrary one-level deep contexts,
- we want to maintain the invariant that for all pairs of relations we consider, $\mathcal{R}^s \subseteq \mathcal{R}^n$, hence we include $\widehat{\mathcal{R}}^s$ in the definition of $\widehat{\mathcal{R}}^n$, and
- if two insensitive bisimilar expressions are thunked, the resulting expressions are sensitive bisimilar; for this reason the proof of Theorem 4.7 requires $\mathsf{fix}(x = \mathsf{fn}\, y \Rightarrow e) \mathrel{\widehat{\mathcal{R}}^s} \mathsf{fix}(x = \mathsf{fn}\, y \Rightarrow f)$ when $e \mathrel{\mathcal{R}^n} f$.

PROPOSITION 4.1. *If $\mathcal{R}$ is an equivalence and $\widehat{\mathcal{R}} \subseteq \mathcal{R}$, then $\mathcal{R}^s$ is a congruence and $\mathcal{R}^n$ is an uneventful congruence.*

PROOF. A variant of the proof in [11, 17]. Show by induction on $C$ that if $e_i \mathrel{\mathcal{R}^s} f_i$ then $C[\vec{e}] \mathrel{\mathcal{R}^s} C[\vec{f}]$. Either $C = \cdot_i$, in which case the result is immediate, or $C = D[\vec{C}]$ and by induction $C_i[\vec{e}] \mathrel{\mathcal{R}^s} C_i[\vec{f}]$, so by definition $C[\vec{e}] = D[\vec{C}[\vec{e}]] \mathrel{\widehat{\mathcal{R}}^s} D[\vec{C}[\vec{f}]] = C[\vec{f}]$. It follows that $\mathcal{R}^s$ is a congruence. The proof that $\mathcal{R}^n$ is an uneventful congruence is similar. □

For any $\mathcal{R}$, its *compatible closure*, $\mathcal{R}^\bullet$, is given by:

$$\frac{e \mathrel{\widehat{\mathcal{R}^\bullet}} e' \mathrel{\mathcal{R}^\circ} e''}{e \mathrel{\mathcal{R}^\bullet} e''}$$

Note that $\mathcal{R}^{\bullet s} \subseteq \mathcal{R}^{\bullet n}$.

This definition of $\mathcal{R}^\bullet$ is specifically designed to facilitate simultaneous inductive proof on syntax (since the definition involves one-level deep contexts) and on reductions (since the definition involves inductive use of $\mathcal{R}^\circ$). This form of induction is precisely what is required to show the desired congruence results.

Its relevant properties are summed up in the following proposition.

PROPOSITION 4.2. *If $\mathcal{R}^\circ$ is a preorder then $\mathcal{R}^\bullet$ is the smallest relation satisfying:*

1. $\mathcal{R}^\bullet \mathcal{R}^\circ \subseteq \mathcal{R}^\bullet$,
2. $\widehat{\mathcal{R}^\bullet} \subseteq \mathcal{R}^\bullet$, *and*
3. $\mathcal{R}^\circ \subseteq \mathcal{R}^\bullet$.

PROOF. A variant of the proof in [11].

First we show that $\mathcal{R}^\bullet$ is reflexive, by showing by structural induction on $e$ that $e \mathrel{\mathcal{R}^{\bullet s}} e$. Find $D[\vec{e}]$ such that $e = D[\vec{e}]$, so by induction $e_i \mathrel{\mathcal{R}^{\bullet s}} e_i$, so by definition of $\widehat{\mathcal{R}}$, $e = D[\vec{e}] \mathrel{\widehat{\mathcal{R}^\bullet}^s} D[\vec{e}] \mathrel{\mathcal{R}^{s\circ}} D[\vec{e}] = e$.

Then we show the required properties:

1. $\mathcal{R}^\bullet \mathcal{R}^\circ \subseteq \widehat{\mathcal{R}^\bullet} \mathcal{R}^\circ \mathcal{R}^\circ \subseteq \widehat{\mathcal{R}^\bullet} \mathcal{R}^\circ \subseteq \mathcal{R}^\bullet$.
2. $\widehat{\mathcal{R}^\bullet} \subseteq \widehat{\mathcal{R}^\bullet} \mathcal{R}^\circ \subseteq \mathcal{R}^\bullet$.
3. $\mathcal{R}^\circ \subseteq \mathcal{R}^\bullet \mathcal{R}^\circ \subseteq \mathcal{R}^\bullet$.

To see that $\mathcal{R}^\bullet$ is the smallest relation satisfying these properties we show that if $\mathcal{S}$ satisfies these properties, then $\widehat{\mathcal{S}} \mathcal{R}^\circ \subseteq \mathcal{S} \mathcal{R}^\circ \subseteq \mathcal{S}$, and so $\mathcal{R}^\bullet \subseteq \mathcal{S}$. □

Since $\widehat{\mathcal{R}^\bullet} \subseteq \mathcal{R}^\bullet$, we know from Proposition 4.1 that if $\mathcal{R}^\bullet$ is an equivalence then $\mathcal{R}^s$ is a congruence and $\mathcal{R}^n$ is an uneventful congruence. However, we can show a stronger result than that, which is that $\mathcal{R}^\bullet$ is closed under substitution of closed values:

PROPOSITION 4.3. *If $\mathcal{R}$ is a preorder then for any $v \mathrel{\mathcal{R}^{\bullet s}} w$:*

1. *if $e \mathrel{\mathcal{R}^{\bullet s}} f$ then $e[v/x] \mathrel{\mathcal{R}^{\bullet s}} f[w/x]$, and*
2. *if $e \mathrel{\mathcal{R}^{\bullet n}} f$ then $e[v/x] \mathrel{\mathcal{R}^{\bullet n}} f[w/x]$.*

PROOF. A variant of the proof in [11, 17]. We shall prove the first part, and the second is similar.

We proceed by induction on $e$.

- If $e = x$ then $x \mathrel{\mathcal{R}^{s\circ}} f$, so $e[v/x] = v \mathrel{\mathcal{R}^{\bullet s}} w \mathrel{\mathcal{R}^{s\circ}} f[w/x]$ so by Proposition 4.2 $e[v/x] \mathrel{\mathcal{R}^{\bullet s}} f[w/x]$.
- If $e = \mathsf{fix}(y = \mathsf{fn}\, z \Rightarrow e_1)$ then we can find a $g_1$ such that $e_1 \mathrel{\mathcal{R}^{\bullet n}} g_1$ and $\mathsf{fix}(y = \mathsf{fn}\, z \Rightarrow g_1) \mathrel{\mathcal{R}^{s\circ}} f$, so by induction $e_1[v/x] \mathrel{\mathcal{R}^{\bullet n}} g_1[w/x]$, so $e[v/x] = \mathsf{fix}(y = \mathsf{fn}\, z \Rightarrow e_1[v/x]) \mathrel{\widehat{\mathcal{R}^\bullet}^s} \mathsf{fix}(y = \mathsf{fn}\, z \Rightarrow g_1[w/x]) \mathrel{\mathcal{R}^{s\circ}} f[w/x]$, so by definition of $\mathcal{R}^\bullet$, $e[v/x] \mathrel{\mathcal{R}^{\bullet s}} f[w/x]$.
- Otherwise, we have $e = D[\vec{e}]$ and $D[\vec{e}][v/x] = D[\vec{e}[v/x]]$, so we can find $\vec{g}$ such that $\vec{e} \mathrel{\mathcal{R}^{\bullet s}} \vec{g}$ and $D[\vec{g}] \mathrel{\mathcal{R}^{s\circ}} f$, so by induction $e_i[v/x] \mathrel{\mathcal{R}^{\bullet s}} f_i[w/x]$, hence $e[v/x] = D[\vec{e}][v/x] = D[\vec{e}[v/x]] \mathrel{\widehat{\mathcal{R}^\bullet}^s} D[\vec{f}[w/x]] = D[\vec{f}][w/x] \mathrel{\mathcal{R}^{s\circ}} f[w/x]$, so by definition of $\mathcal{R}^\bullet$, $e[v/x] \mathrel{\mathcal{R}^{\bullet s}} f[w/x]$. □

Our proof strategy is to show that $\approx^\circ$ and $\approx^\bullet$ coincide. Since $\approx^\circ \subseteq \approx^\bullet$, this amounts to showing that $\approx^\bullet \subseteq \approx^\circ$, which we do by proving that $\approx^\bullet$, when restricted to programs, is a hereditary simulation.

PROPOSITION 4.4. *When restricted to closed expressions of $\mu CML^+$, $\approx^\bullet$ is a hereditary simulation.*

PROOF. We have to show that $\approx^{\bullet s}$ is structure-preserving, and that the diagrams for a hereditary simulation can be completed.

Showing that $\approx^{\bullet s}$ is structure preserving is a routine structural induction.

If:

$$\begin{tikzcd} e \arrow[d, "l_1"'] & \approx^{\bullet n} & f \\ e' & & \end{tikzcd}$$

then we proceed by induction on $e$ to show that we can complete the diagram as:

$$\begin{tikzcd} e \arrow[d, "l_1"'] & \approx^{\bullet n} & f \arrow[d, Rightarrow, "\hat{l}_2"'] \\ e' & \approx^{\bullet n} & f' \end{tikzcd}$$

where $l_1 \approx^{\bullet sl} l_2$, and similarly for $\approx^{\bullet s}$. We shall show three of the more interesting cases, the others are similar but more routine:

- if we have:

$$\begin{tikzcd} e \arrow[r, equal] \arrow[d, "\tau"'] & \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 \arrow[d, "\tau"'] \widehat{\approx^\bullet}^n \mathsf{let}\, x = g_1 \,\mathsf{in}\, g_2 \approx^n & f \\ e' \arrow[r, equal] & e'_1 \parallel e_2[v/x] & \end{tikzcd}$$

  where $e_i \approx^{\bullet n} g_i$ and $e_1 \xrightarrow{\surd v} e'_1$, then by induction $g_1 \xRightarrow{\surd w} g'_1$, $v \approx^{\bullet s} w$ and $e'_1 \approx^{\bullet n} g'_1$, so using Proposition 4.3, we have:

$$\begin{tikzcd} e \arrow[r, equal] \arrow[d, "\tau"'] & \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 \arrow[d, "\tau"'] & \widehat{\approx^\bullet}^n & \mathsf{let}\, x = g_1 \,\mathsf{in}\, g_2 \arrow[d, Rightarrow, "\tau"'] & \approx^n & f \arrow[d, Rightarrow, "\varepsilon"'] \\ e' \arrow[r, equal] & e'_1 \parallel e_2[v/x] & \approx^{\bullet n} & g'_1 \parallel g_2[w/x] & \approx^n & f' \end{tikzcd}$$



- if we have:

$$\begin{tikzcd} e \arrow[r, equal] \arrow[d, "\tau"'] & e_1\, e_2 \arrow[d, "\tau"'] & \widehat{\approx^\bullet}^n & g_1\, g_2 & \approx^n & f \\ e' \arrow[r, equal] & e'_1 \parallel \mathsf{let}\, y = e_2 \,\mathsf{in}\, e_3[v/x] & & & & \end{tikzcd}$$

  where $e_i \approx^{\bullet n} g_i$, $e_1 \xrightarrow{\surd v} e'_1$, and $v = \mathsf{fix}(x = \mathsf{fn}\, y \Rightarrow e_3)$ then by induction $g_1 \xRightarrow{\surd w} g'_1$, $v \approx^{\bullet s} w$, up to α-conversion $w = \mathsf{fix}(x = \mathsf{fn}\, y \Rightarrow g_3)$, and $e'_1 \approx^{\bullet n} g'_1$. Then by the definition of $\approx^\bullet$, we can find an $v' = \mathsf{fix}(x = \mathsf{fn}\, y \Rightarrow h_3)$ such that $e_3 \approx^{\bullet n} h_3$ and $v' \approx^s w$, so by Proposition 4.3, $e_3[v/x] \approx^{\bullet n} h_3[v'/x] \approx^{n\circ} v'\, y \approx^{n\circ} w\, y \approx^{n\circ} g_3[w/x]$, and so:

$$\begin{tikzcd} e \arrow[r, equal] \arrow[d, "\tau"'] & e_1\, e_2 \arrow[d, "\tau"'] & \widehat{\approx^\bullet}^n & g_1\, g_2 \arrow[d, Rightarrow, "\tau"'] & \approx^n & f \arrow[d, Rightarrow, "\varepsilon"'] \\ e' \arrow[r, equal] & e'_1 \parallel \mathsf{let}\, y = e_2 \,\mathsf{in}\, e_3[v/x] & \approx^{\bullet n} & g'_1 \parallel \mathsf{let}\, y = g_2 \,\mathsf{in}\, g_3[w/x] & \approx^n & f' \end{tikzcd}$$

- if we have:

$$\begin{tikzcd} e \arrow[r, equal] \arrow[d, "\surd e"'] & \mathsf{fix}(x = \mathsf{fn}\, y \Rightarrow e_1) \arrow[d, "\surd e"'] \widehat{\approx^\bullet}^n \mathsf{fix}(x = \mathsf{fn}\, y \Rightarrow g_1) \approx^n & f \\ e' \arrow[r, equal] & \Lambda & \end{tikzcd}$$

  where $e_1 \approx^{\bullet n} g_1$ then let $v = \mathsf{fix}(x = \mathsf{fn}\, y \Rightarrow g_1)$, so:

$$\begin{tikzcd} e \arrow[r, equal] \arrow[d, "\surd e"'] & \mathsf{fix}(x = \mathsf{fn}\, y \Rightarrow e_1) \arrow[d, "\surd e"'] & \widehat{\approx^\bullet}^n & \mathsf{fix}(x = \mathsf{fn}\, y \Rightarrow g_1) \arrow[d, Rightarrow, "\surd v"'] & \approx^n & f \arrow[d, Rightarrow, "\surd w"'] \\ e' \arrow[r, equal] & \Lambda & \approx^{\bullet n} & \Lambda & \approx^n & f' \end{tikzcd}$$

  and $e \approx^{\bullet s} v \approx^s w$.

Thus $\approx^\bullet$ is a hereditary simulation. □

We now have that $\approx^\bullet$ is a simulation, and we would like to show that it is a bisimulation, for which it suffices to show that $\approx^\bullet$ is symmetric. Unfortunately, this is not easy to prove directly, and so we use a result of Howe's [18] (pointed out to the authors by Andrew Pitts) which allows us to show that $\approx^{\bullet *}$ is symmetric.

PROPOSITION 4.5. *If $\mathcal{R}$ is an equivalence then $\mathcal{R}^{\bullet *}$ is symmetric.*

PROOF. A variant of the proof in [18].

It suffices to show that if $e \mathcal{R}^{\bullet s} f$ then $f \mathcal{R}^{\bullet s*} e$, and that if $e \mathcal{R}^{\bullet n} f$ then $f \mathcal{R}^{\bullet n*} e$, which we show by induction on $e$. If $e \mathcal{R}^{\bullet s} f$, then either:

- $e = D[\vec{e}]\ \widehat{\mathcal{R}^{\bullet}}^{s}\ D[\vec{f}]\ \mathcal{R}^{s\circ} f$ and $e_i \mathcal{R}^{\bullet s} f_i$, so by induction $f_i \mathcal{R}^{\bullet s*} e_i$, so $f\ \widehat{\mathcal{R}}^{s}\ D[\vec{f}] D\ \widehat{\mathcal{R}}^{s*} [\vec{e}] = e$, or
- $e = \mathsf{fix}(x = \mathsf{fn}\ y \Rightarrow e')\ \widehat{\mathcal{R}^{\bullet}}^{s}\ \mathsf{fix}(x = \mathsf{fn}\ y \Rightarrow f')\ \mathcal{R}^{s\circ} f$ and $e' \mathcal{R}^{\bullet n} f'$, so by induction $f' \mathcal{R}^{\bullet n*} e'$, so $f\ \widehat{\mathcal{R}}^{s}\ \mathsf{fix}(x = \mathsf{fn}\ y \Rightarrow f')\ \mathcal{R}^{\bullet s*}\ \mathsf{fix}(x = \mathsf{fn}\ y \Rightarrow e') = e$.

The proof for $\mathcal{R}^n$ is similar. □

We can use this result to show that $\approx^{\bullet *}$ is a bisimulation.

PROPOSITION 4.6. *When restricted to closed expressions of $\mu$CML$^+$, $\approx^{\bullet *}$ is a hereditary bisimulation.*

PROOF. By Proposition 4.4, $\approx^{\bullet}$ is a hereditary simulation, and so $\approx^{\bullet *}$ is a hereditary simulation. By Proposition 4.5, $\approx^{\bullet}$ is symmetric, and so $\approx^{\bullet}$ is a hereditary bisimulation. □

This gives us the result we set out to prove.

THEOREM 4.7. *$\approx^s$ is a congruence, and $\approx^n$ is an uneventful congruence.*

PROOF. From Proposition 4.6, $\approx^{\bullet}$ is a hereditary bisimulation, so $\approx^{\bullet} \subseteq \approx^{\circ}$, and by Proposition 4.2 $\approx^{\circ} \subseteq \approx^{\bullet}$, so $\approx^{\bullet}$ and $\approx^{\circ}$ are the same relation. Since $\widehat{\approx^{\bullet}} \subseteq \approx^{\bullet}$, we have the desired result by Proposition 4.1. □

## 5 Properties of Weak Bisimulation

In this section, we show some results about program equivalence up to hereditary weak bisimulation. Some of these equivalences are easy to show, but some are trickier, and require properties about the transition systems generated by $\mu$CML$^+$. Although much remains to be done on elaborating the algebraic theory of $\mu$CML programs we hope that the results in this section indicate that this equivalence can form the basis of a useful theory which generalises those associated with process algebras and functional programming.

We have given an operational semantics to $\mu$CML by extending it with new constructs, most of which correspond to constructs found in standard process algebras. These include a choice operator $\oplus$, a parallel operator $\|$ and suitable versions of input and output prefixing, [20]. The prefixes in $\mu$CML$^{cv}$ have a slightly unusual syntax—their equivalents in CCS are given as:

| *CCS prefix* | *$\mu$CML$^{cv}$ equivalent* |
|---|---|
| $k?x.P$ | $k? \Rightarrow \mathsf{fn}\ x \Rightarrow P$ |
| $k!v.P$ | $k!v \Rightarrow \mathsf{fn}\ x \Rightarrow P$ |
| $\tau.P$ | $\mathbf{A}() \Rightarrow \mathsf{fn}\ x \Rightarrow P$ |



We now examine the extent to which $\oplus$ and $\|$ act like choice and parallel operators from a process algebras

We can find bisimulations for the following (and hence they are sensitive bisimilar):

$$\Lambda \,\|\, e \sim^1 e$$
$$(e_1 \,\|\, e_2) \,\|\, e_3 \sim^1 e_1 \,\|\, (e_2 \,\|\, e_3)$$
$$(e_1 \,\|\, e_2) \,\|\, e_3 \sim^1 (e_2 \,\|\, e_1) \,\|\, e_3$$

Thus $\|$ satisfies many of the standard laws associated with a parallel operator in a process algebra. However it is not in general symmetric because of its interaction with the production of values:

$$v \,\|\, e \sim^1 e$$

For example:

$$1 \,\|\, \Lambda \sim^1 \Lambda \qquad \Lambda \,\|\, 1 \sim^1 1$$

This means that we can view the parallel composition of processes as being of the form:

$$(\|_i\, e_i) \,\|\, f$$

where the order of the $e_i$ is unimportant. Note that it *is* important which is the right-most expression in a parallel composition, since it is the main thread of computation, and so can return a value, which none of the other expressions can.

The choice operator of $\mu$CML$^+$ also satisfies the expected laws from process algebras, those of a commutative monoid, although it can only be applied to guarded expressions:

$$\Lambda \oplus ge \sim^1 ge$$
$$(ge_1 \oplus ge_2) \oplus ge_3 \sim^1 ge_1 \oplus (ge_2 \oplus ge_3)$$
$$ge_1 \oplus ge_2 \sim^1 ge_2 \oplus ge_1$$

This means that we can view the sum of guarded expressions as being of the form:

$$\bigoplus_i ge_i$$

where the order of the $ge_i$ is unimportant.

In fact guarded expressions can be viewed in a manner quite similar to the *sum forms* used in the development of the algebraic theory of CCS, [20]. We can find bisimulations for the following (and hence they are sensitive bisimilar):

$$\begin{aligned}(ge_1 \oplus ge_2) \Rightarrow v &\sim^1 (ge_1 \Rightarrow v) \oplus (ge_2 \Rightarrow v)\\ ge \Rightarrow \mathsf{fn}\, x \Rightarrow x &\approx^s ge\\ \mathbf{A} v &\approx^s \mathbf{A}() \Rightarrow \mathsf{fn}\, x \Rightarrow v\end{aligned}$$

From this, we can show, by structural induction on that all guarded expressions are of a given form:

$$ge \approx^s \bigoplus_i ge_i \Rightarrow v_i$$

where each $ge_i$ is either $k_i!v_i$, $k_i?$ or $\mathbf{A}()$. From this and:

$$c\, v \approx^1 \delta(c, v)$$

we can show that all values $\vdash v : A\,\mathsf{event}$ are of the form:

$$v \approx^n \mathsf{choose}[\mathsf{wrap}(e_1, v_1), \ldots, \mathsf{wrap}(e_n, v_n)]$$

where $e_n$ is either $\mathsf{transmit}(k_i, v_i)$, $\mathsf{receive}\, k_i$, or $\mathsf{always}()$.

We could continue in this manner emulating the algebraic theory of CCS, for example with expansion theorems for guarded expressions or values of event type. However we leave this for future work.

We now turn our attention to $\mu$CML viewed as a functional language. One would not expect $\beta$-reduction in its full generality in a language with side-effects such as $\mu$CML but we do obtain an appropriate call-by-value version:

$$(\mathsf{fn}\, y \Rightarrow e)\, v \approx^1 e[v/y]$$

We also have expected laws such as:

$$\begin{aligned}\mathsf{fst}(e, v) &\approx^1 e\\ \mathsf{snd}(v, e) &\approx^1 e\\ (\mathsf{fix}(x = \mathsf{fn}\, y \Rightarrow e))\, v &\approx^1 e[\mathsf{fix}(x = \mathsf{fn}\, y \Rightarrow e)/x][v/y]\\ \mathsf{let}\, x = v\, \mathsf{in}\, e &\approx^1 e[v/x]\\ \mathsf{let}\, y = (\mathsf{let}\, x = e\, \mathsf{in}\, f)\, \mathsf{in}\, g &\approx^1 \mathsf{let}\, x = e\, \mathsf{in}\, (\mathsf{let}\, y = f\, \mathsf{in}\, g) \quad \text{where } x \notin fv(g)\end{aligned}$$

The last two equations are of particular interest, since they are exactly the left unit and associativity axioms of Moggi's [23] monadic metalanguage. The right unit equation:

$$\mathsf{let}\, x = e\, \mathsf{in}\, x \approx^n e$$

is not so simple to show, and indeed if $e$ were an arbitrary labelled transition



system then it would not be true, as can be seen by:

$$\begin{tikzcd} & e \arrow[dl, "k!()"'] \arrow[dr, "\surd v"] & & \not\approx^n & & \mathsf{let}\, x = e\, \mathsf{in}\, x \arrow[dl, "k!()"'] \arrow[dr, "\tau"] & \\ \Lambda & & \Lambda & & \Lambda & & \Lambda \parallel v \arrow[d, "\surd v"'] \\ & & & & & & \Lambda \end{tikzcd}$$

(This is the same example which makes *SKIP* not act as a right unit for ; in CSP [15] and **exit** not act as a right unit for $\gg$ in LOTOS [1].) Fortunately, we can show that our operational semantics for $\mu$CML satisfies four properties which allow us to show the right unit equation.

A labelled transition system is *single-valued* iff:

$$\text{if } e \xrightarrow{\surd v} e' \text{ then } e' \not\xrightarrow{\surd w}$$

It is *value deterministic* iff:

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

It is *forward commutative* iff:

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

It is *backward commutative* iff:

$$\text{if}\quad \begin{array}{ccc} e & \xrightarrow{\surd v} & e' \\ & & \Big\downarrow{\scriptstyle\alpha} \\ & & e''' \end{array} \quad\text{then}\quad \begin{array}{ccc} e & \xrightarrow{\surd v} & e' \\ {\scriptstyle\alpha}\Big\downarrow & & \Big\downarrow{\scriptstyle\alpha} \\ e'' & \xrightarrow{\surd v^{\circ}} & e''' \end{array}$$

Note in particular that LOTOS and CSP do not satisfy forward commutativity, which is why their sequential composition operators do not have a right unit. However, $\mu$CML does satisfy these conditions.

PROPOSITION 5.1. *$\mu$CML satisfies single-valuedness, value determinacy, forward commutativity and backward commutativity.*

PROOF. A routine induction on syntax. □

The important property which such lts's satisfy is the following, where we assume the existence of the operator $\|$.

PROPOSITION 5.2. *In any single-valued, value deterministic, forward commutative, backward commutative lts, if $e \xrightarrow{\surd v} e'$ then $e \approx^1 e' \| v$.*

PROOF. Use the properties of the lts to establish that the following is a first-order weak bisimulation:

$$\{(e, e' \| v) \mid e \xrightarrow{\surd v} e'\} \cup \{(e', e' \| \Lambda) \mid e \xrightarrow{\surd v} e'\}$$

The result follows. □

As a corollary to this proposition, it is routine to show that the following is a first-order weak bisimulation:

$$\{(e, \mathsf{let}\, x = e\, \mathsf{in}\, x)\} \cup \approx^1$$

So we have the right unit equation we were looking for:

$$e \approx^1 \mathsf{let}\, x = e\, \mathsf{in}\, x$$

These equations enable us to define a categorical model for $\mu$CML where:

- objects are types,
- morphisms between $A$ and $B$ are typed expressions with one free variable $x : A \vdash e : B$, viewed up to weak bisimulation,
- the identity morphism is $x : A \vdash x : A$, and
- composition is $(x : A \vdash e : B); (y : B \vdash f : C) = (x : A \vdash \mathsf{let}\, y = e\, \mathsf{in}\, f : C)$.



The equations for weak bisimulation discussed above show that morphism composition is associative and has the identity as both a left unit and right unit. Thus $\mu$CML forms a category.

Again we leave the investigation of the properties of this category to future work but we should point out that so far we have been unable to cast it as an instance of general categorical framework of [23].

## 6 Comparing $\mu$CML$^+$ and $\lambda_{cv}$

In section 2 we presented the operational semantics of a subset of CML, as a labelled transition system, in order that we might investigate its behavioural properties. In this section we shall make formal connection between this semantics and the reduction semantics for $\lambda_{cv}$ presented in [30]. We have not considered $\lambda_{cv}$ in its entirety and so the comparison will be confined to the common subset, namely $\mu$CML$^{cv}$. We first reproduce, as faithfully as possible, the reduction semantics of Reppy as it applies to $\mu$CML. From this reduction semantics we then derive a labelled transition system for $\mu$CML expressions and our main theorem states that this labelled transition system (up to first-order weak bisimulation) is the same as ours. In fact the more technical results we derive connecting the two semantics would support a much closer relationship but expressing it would involve developing yet another bisimulation based equivalence.

Before presenting the operational semantics and our main theorem we clarify the differences between $\lambda_{cv}$ and $\mu$CML$^{cv}$:

- We do not consider the $\lambda_{cv}$ constructs `guard` and `wrapAbort`. We conjecture that the operational semantics of $\mu$CML would need to be considerably altered to cope with translating these constructs.
- We omit the $\lambda_{cv}$ construct `chan x in e` since we cannot encode unique channel name generation in $\mu$CML. It should not be difficult to add unique channel name generation to $\mu$CML using operational rules à la $\pi$-calculus, although this would require using a bisimulation similar to Sangiorgi's [31] context bisimulation for the higher-order $\pi$-calculus.
- We have added recursive function types to $\mu$CML$^{cv}$ because in [30] recursion is encoded using process creation and unique channel name generation.
- In $\lambda_{cv}$, constant functions such as `wrap` are values, where in $\mu$CML they have to be coded as $(\mathsf{fn}\, x \Rightarrow \mathsf{wrap}\, x)$. This restriction has no effect on the expressive power of $\mu$CML, and makes it simpler to reason about the operational semantics, since any value of type $A \to B$ must be of the form $\mathsf{fix}(x = \mathsf{fn}\, y \Rightarrow e)$.

We now present Reppy's reduction semantics for $\mu$CML$^{cv}$. In [30] this is represented by a transition relation between multi-sets of $\mu$CML$^{cv}$, or more generally

$\lambda_{cv}$ expressions. Instead of multi-sets we use *configurations* of $\mu\text{CML}^{cv}$ expressions given by the grammar:

$$C \in Conf ::= e \mid C \parallel C \mid \Lambda$$

Note that configurations are restricted forms of $\mu\text{CML}^{+}$ expressions. This will facilitate the comparison between the two semantics since it can be carried out for configurations rather than $\mu$CML expressions.

The semantics of [30] is expressed as a reduction relation $\Longrightarrow$ between configurations and reductions have four independent sources. The first involves a sequential reduction within an individual $\mu$CML expression and this in turn is defined using another reduction relation $\longmapsto$; the second is the spawning of new *computation threads* which results in an increase in the number of components of the configuration; the third is communication between two expressions and the last is required to handle the always construct. We need notation for each of these and we consider them in turn.

The operational rules for sequential reduction are defined *in context* in the style of Wright and Felleisen [33], and the contexts that permit reduction are given by the following grammar:

$$E ::= [\cdot] \mid E\,e \mid v\,E \mid c\,E \mid (E,e) \mid (v,E) \mid \mathsf{let}\,x = E\,\mathsf{in}\,e \mid \mathsf{if}\,E\,\mathsf{then}\,e\,\mathsf{else}\,e$$

The relation $\longmapsto$ is defined to be the least relation satisfying the following rules:

$$\begin{array}{rcll}
E[c\,v] & \longmapsto & E[\delta(c\,v)] \qquad (c \notin \{\mathsf{spawn}, \mathsf{sync}\}) & \underline{\text{const}} \\
E[(\mathsf{fix}(x = \mathsf{fn}\,y \Rightarrow e))\,v] & \longmapsto & E[e[\mathsf{fix}(x = \mathsf{fn}\,y \Rightarrow e)/x][v/y]] & \underline{\text{beta}} \\
E[\mathsf{let}\,x = v\,\mathsf{in}\,e] & \longmapsto & E[e[v/x]] & \underline{\text{let}} \\
E[(v,w)] & \longmapsto & E[\langle v,w \rangle] & \underline{\text{pair}}
\end{array}$$

Here each rule corresponds to a basic computation step in a sequential call-by-value language. We should point out that the last rule does not appear in [30], it is implicit in Reppy's statement "the syntactic class of the term $(v_1, v_2)$ is either *Exp* or *Val*; this ambiguity is resolved in favour of *Val*." We have made the grammar unambiguous, and have added an explicit reduction rule for resolving ambiguity.

Note that the definition of $\longmapsto$ is not compositional: the reductions of an expression are not defined in terms of the reductions of its sub-expressions. The following Lemma will be useful in later proofs and shows that we can recover compositionality.

LEMMA 6.1. *If $e \longmapsto e'$ then $E[e] \longmapsto E[e']$.*

PROOF. By examination of the proof of the transition $e \longmapsto e'$. □

To capture reductions which involve communication it is necessary to define a notion of when two guarded expressions may give rise to a communication. For



$$\frac{}{k!v \stackrel{k}{\bowtie} k?\ \mathbf{with}\ ((),v)} \qquad \frac{ge \stackrel{k}{\bowtie} ge'\ \mathbf{with}\ (e,e')}{ge \stackrel{k}{\bowtie} ge' \Rightarrow v\ \mathbf{with}\ (e, v\,e')}$$

$$\frac{ge \stackrel{k}{\bowtie} ge'\ \mathbf{with}\ (e,e')}{ge \stackrel{k}{\bowtie} ge' \oplus ge''\ \mathbf{with}\ (e,e')} \qquad \frac{ge \stackrel{k}{\bowtie} ge''\ \mathbf{with}\ (e,e'')}{ge \stackrel{k}{\bowtie} ge' \oplus ge''\ \mathbf{with}\ (e,e'')}$$

$$\frac{ge \stackrel{k}{\bowtie} ge'\ \mathbf{with}\ (e,e')}{ge' \stackrel{k}{\bowtie} ge\ \mathbf{with}\ (e',e)}$$

FIGURE 5A. The rules for matching events

$$\frac{}{\mathbf{A}v \rhd v} \qquad \frac{ge \rhd e}{ge \Rightarrow v \rhd v\,e} \qquad \frac{ge \rhd e}{ge \oplus ge' \rhd e} \qquad \frac{ge' \rhd e'}{ge \oplus ge' \rhd e'}$$

FIGURE 5B. The rules for immediate evaluation of events

any $k$ the relation:

$$ge \stackrel{k}{\bowtie} ge'\ \mathbf{with}\ (e,e')$$

read as "$ge$ matches $ge'$ on $k$ with result $(e,e')$" is defined to be the least relation satisfying the rules in Figure 5a. Intuitively this means that two concurrent threads $e_1, e_2$ of the form $e_1 = E_1[\mathsf{sync}[ge]], e_2 = E_2[\mathsf{sync}[ge']]$ may communicate in one step on the channel $k$ with $E_1[e]$ and $E_2[e']$ being the result of this communication.

To handle reductions caused by always we need to formalise when guarded expressions such as $\mathbf{A}v$ can immediately return values. This is given by Reppy's relation $ge \rhd e$, is defined in Figure 5b.

We can now formally present the reduction relation $\Longrightarrow$ between configurations. It is defined to be the least relation satisfying the rules:

$$\frac{e_i \longmapsto e_i'}{(e_1 \parallel \cdots \parallel e_i \parallel \cdots \parallel e_n) \Longrightarrow (e_1 \parallel \cdots \parallel e_i' \parallel \cdots \parallel e_n)} \quad \underline{\text{seq}}$$

$$\frac{}{(e_1 \parallel \cdots \parallel E[\mathsf{spawn}\,v] \parallel \cdots \parallel e_n) \Longrightarrow (e_1 \parallel \cdots \parallel v() \parallel E[()] \parallel \cdots \parallel e_n)} \quad \underline{\text{spawn}}$$

$$\frac{ge \stackrel{k}{\bowtie} ge'\ \mathbf{with}\ (e,e')}{\begin{array}{c}(e_1 \parallel \cdots \parallel E[\mathsf{sync}[ge]] \parallel \cdots \parallel E'[\mathsf{sync}[ge']] \parallel \cdots \parallel e_n) \\ \Longrightarrow (e_1 \parallel \cdots \parallel E[e] \parallel \cdots \parallel E'[e'] \parallel \cdots \parallel e_n)\end{array}} \quad \underline{\text{comm}}$$

$$\frac{ge \rhd e}{(e_1 \parallel \cdots \parallel E[\mathsf{sync}[ge]] \parallel \cdots \parallel e_n) \Longrightarrow (e_1 \parallel \cdots \parallel E[e] \parallel \cdots \parallel e_n)} \quad \underline{\text{eval}}$$

This completes our exposition of Reppy's semantics as it applies to $\mu\text{CML}^{cv}$, which for convenience we call the $\mu\text{CML}^{cv}$ semantics. We refer to that in Section 2 as the $\mu$CML$^+$ semantics and we now compare them. In order to do this, we extract a labelled transition system from the $\mu$CML$^{cv}$ semantics by defining:

$C \overset{\tau}{\longmapsto} C'$ iff $C \Longrightarrow C'$

$C \overset{\surd v}{\longmapsto} C'$ iff $C = C'' \parallel v$ and $C' = C'' \parallel \Lambda$ (up to $\parallel$ associativity and $\Lambda$ left unit)

$C \overset{k!v}{\longmapsto} C'$ iff $C \parallel k? \Longrightarrow C' \parallel v$

$C \overset{k?x}{\longmapsto} C'$ iff $C \parallel k!x \Longrightarrow C' \parallel ()$

We will then show that this labelled transition system is weakly bisimilar to the $\mu$CML$^+$ lts:

THEOREM 6.2. *The $\mu$CML$^{cv}$ semantics of a configuration is weakly bisimilar to its $\mu$CML$^+$ semantics.*

The remainder of this section is devoted to proving this result. Although the style of presentation of these two semantics are very different the resulting relations are very similar and there are essentially only two sources for the differences. The first is that certain reductions in $\mu$CML$^{cv}$, when modelled in the $\mu$CML$^+$ semantics, require in addition some 'housekeeping' reductions. A typical example is the reduction:

$$(\mathsf{fn}\, x \Rightarrow e)v \longmapsto e[v/x].$$

In $\mu$CML$^+$ this requires two reductions:

$$(\mathsf{fn}\, x \Rightarrow e)v \xrightarrow{\tau} \mathsf{let}\, x = v\, \mathsf{in}\, e \xrightarrow{\tau} e[v/x]$$

This problem is handled by identifying the set of 'housekeeping' reductions, such as the second reduction above, within the $\mu$CML$^+$ semantics . These turn out to be very simple and we can work with 'housekeeping normal forms' in which no further housekeeping reductions can be made.

The second divergence between the semantics concerns the treatment of $\mathsf{spawn}$; expressions in $\mu$CML$^+$ may spawn new processes which give rise to parallel processes occurring as sub-terms of the expression. For example, the reductions of $(\mathsf{spawn}\, v, e)$ in $\mu$CML$^+$ and $\mu$CML$^{cv}$ are:

$$(\mathsf{spawn}\, v, e) \xrightarrow{\tau} (\Lambda \parallel v\,() \parallel (), e)$$
$$(\mathsf{spawn}\, v, e) \overset{\tau}{\longmapsto} v\,() \parallel ((), e)$$

This difference is handled by working with the $\mu$CML$^{cv}$ semantics up to a syntactically defined equivalence; this equivalence is contained in strong bisimulation equivalence and it also preserves housekeeping reductions.

We now explain in some more detail these technical developments; most of the associated proofs are relegated to an Appendix. House-keeping reductions are ones derived using the rules:



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

$$\frac{e \xrightarrow{\surd v} e'}{e\, f \xrightarrow{\tau} e' \parallel \mathsf{let}\, y = f\, \mathsf{in}\, g[v/x]} [v = \mathsf{fix}(x = \mathsf{fn}\, y \Rightarrow g)]$$

We shall write $e \xrightarrow{\tau_H} e'$ whenever $e \xrightarrow{\tau} e'$ is a housekeeping reduction.

It is routine to verify that the housekeeping moves are 'semantically unimportant', as is captured by the next proposition:

PROPOSITION 6.3. *If $e \xrightarrow{\tau_H} e'$ then $e \approx^1 e'$.*

PROOF. Construct a weak bisimulation for each case. □

Moreover, we can show a confluence result for the $\mu$CML$^+$ semantics about housekeeping moves:

PROPOSITION 6.4. *If*

$$\begin{tikzcd} e \arrow[r, "\tau_H^*"] \arrow[d, "l"'] & e' \\ e'' & \end{tikzcd}$$

*then*

$$\begin{tikzcd} e \arrow[r, "\tau_H^*"] \arrow[d, "l"'] & e' \arrow[d, "\hat{l}"] \\ e'' \arrow[r, "\tau_H^{*\circ}"] & e''' \end{tikzcd}$$

PROOF. First show by induction on $ge$ that $ge \overset{\tau_H}{\not\longrightarrow}$. Then prove by induction on $e$, using forward commutativity, that if $e \xrightarrow{\tau_H} e'$ and $e \xrightarrow{l} e''$ are distinct reductions then we can find $e'''$ such that $e' \xrightarrow{l} e'''$ and $e'' \xrightarrow{\tau_H} e'''$. The result follows. □

Call a term 'tidy' if it has no housekeeping reductions. Then we can show that every $\mu$CML$^+$ term has a unique tidy normal form.

PROPOSITION 6.5. *For any $\mu$CML$^+$ term $e$ there is a unique tidy $e'$ such that $e \xrightarrow{\tau_H}{}^* e'$.*

PROOF. Show by induction on $e$ that there is some tidy $e'$ such that $e \xrightarrow{\tau_H}{}^* e'$. From Proposition 6.4, this $e'$ is unique. □

We now turn our attention to the syntactic equivalence used to handle the different treatments of $\mathsf{spawn}$. In order to define the equivalence $\equiv$ it is convenient to introduce reduction contexts for $\mu$CML$^+$, equivalent to those for $\mu$CML$^{cv}$:

$$E^+ ::= [\cdot] \mid E^+\, e \mid c\, E^+ \mid (E^+, e) \mid \mathsf{let}\, x = E^+\, \mathsf{in}\, e \mid \mathsf{if}\, E^+\, \mathsf{then}\, e\, \mathsf{else}\, e \mid E^+ \parallel e \mid e \parallel E^+$$

In the Appendix we show that these satisfy the natural properties one would expect of reduction contexts. Let $\equiv$ be the smallest equivalence given by equivalence given by:

$$\frac{}{E^+[\Lambda \parallel e] \equiv E^+[e]} \qquad \frac{}{E_1^+[E_2^+[e \parallel f]] \equiv E_1^+[e \parallel E_2^+[f]]}$$

The equivalence $\equiv$ is a strong first-order bisimulation which respects housekeeping, that is a relation $\mathcal{R}$ where we can complete the diagram:

$$\begin{tikzcd} e_1 \arrow[d, "\tau_H"'] & \mathcal{R} & e_2 \\ e_1' & & \end{tikzcd} \quad \text{as} \quad \begin{tikzcd} e_1 \arrow[d, "\tau_H"'] & \mathcal{R} & e_2 \arrow[d, "\tau_H"'] \\ e_1' & \mathcal{R} & e_2' \end{tikzcd}$$

and similarly for $\mathcal{R}^{-1}$.

PROPOSITION 6.6. *$\equiv$ is a strong first-order bisimulation which respects housekeeping.*

PROOF. See the Appendix. □

We can also show a very strong correspondence between reductions of $\mu$CML$^{cv}$ configurations, and their tidy normal forms.

PROPOSITION 6.7. *If $C \xrightarrow{\tau_H}{}^* e$ and $e$ is tidy, then the following diagrams can be completed:*

$$\begin{tikzcd} C \arrow[r, "\tau_H^*"] \arrow[d, maps to, "l"'] & e \\ C' & \end{tikzcd} \quad \text{as} \quad \begin{tikzcd} C \arrow[r, "\tau_H^*"] \arrow[d, maps to, "l"'] & e \arrow[d, "l"'] \\ C' \arrow[r, "\tau_H^{*\circ}"] & \equiv e' \end{tikzcd}$$

*and:*

$$\begin{tikzcd} C \arrow[r, "\tau_H^*"] & e \arrow[d, "l"'] \\ & e' \end{tikzcd} \quad \text{as} \quad \begin{tikzcd} C \arrow[r, "\tau_H^*"] \arrow[d, maps to, "l"'] & e \arrow[d, "l"'] \\ C' \arrow[r, "\tau_H^{*\circ}"] & \equiv e' \end{tikzcd}$$

PROOF. See the Appendix. □

With these technical results we can now prove the main result showing the correspondence between the two semantics:

THEOREM 6.8. *The $\mu$CML$^{cv}$ semantics of a configuration is weakly bisimilar to its $\mu$CML$^+$ semantics.*

PROOF. Intuitively we know, from Proposition 6.3, that $\mu$CML$^+$ expressions are semantically equivalent to their tidy forms, and Proposition 6.7 can be used



to transform $\mu$CML$^{cv}$ moves from an expression into $\mu$CML$^+$ moves of its tidy form up to $\equiv$, and vice-versa. Formally we show that $\xrightarrow{\tau_H}{}^* \equiv \xleftarrow{\tau_H}{}^*$ is a weak bisimulation by completing the diagram:

$$\begin{tikzcd} C_1 \arrow[r, "\tau_H^*"] \arrow[d, maps to, "l"'] & e_1 \arrow[r, equal] & f_1 & g_1 \arrow[l, "\tau_H^*"'] \\ C_2 & & & \end{tikzcd}$$

by using Proposition 6.5 to find $e_1$'s tidy form $e_2$, and then using Propositions 6.4, 6.6 and 6.7 to show:

$$\begin{tikzcd} C_1 \arrow[r, "\tau_H^*"] \arrow[d, maps to, "l"'] & e_1 \arrow[r, equal] \arrow[d, "\tau_H^*"'] & f_1 \arrow[d, "\tau_H^*"'] & g_1 \arrow[l, "\tau_H^*"'] \arrow[d, "\tau_H^*"'] \\ C_2 \arrow[d, "\tau_H^{*\circ}"'] & e_2 \arrow[r, equal] \arrow[d, "l"'] & f_2 \arrow[d, "l"'] & f_2 \arrow[l, "\tau_H^*"'] \arrow[d, "l"'] \\ e_4 \arrow[r, equal] & e_3 \arrow[r, equal] & f_3 & f_3 \arrow[l, "\tau_H^{*\circ}"'] \end{tikzcd}$$

and by completing the diagram:

$$\begin{tikzcd} C_1 \arrow[r, "\tau_H^*"] & e_1 \arrow[r, equal] & f_1 & g_1 \arrow[l, "\tau_H^*"'] \arrow[d, "l"'] \\ & & & g_2 \end{tikzcd}$$

by using Proposition 6.5 to find $e_1$'s tidy form $e_3$ and then using Propositions

6.4, 6.6 and 6.7 to show:

$$
\begin{tikzcd}
C_1 \arrow[rr, "\tau_H^*"] \arrow[dd, "\hat{l}"'] & & e_1 \arrow[r, phantom, "\equiv"] \arrow[dl, "\tau_H^*"'] \arrow[d, "\hat{l}"'] & f_1 \arrow[d, "\hat{l}"'] & g_1 \arrow[l, "\tau_H^*"'] \arrow[d, "l"'] \\
 & e_3 \arrow[dr, "\hat{l}"] & e_2 \arrow[r, phantom, "\equiv"] \arrow[d, "\tau_H^{*\circ}"'] & f_2 \arrow[d, "\tau_H^{*\circ}"'] & g_2 \arrow[l, "\tau_H^{*\circ}"'] \\
C_2 \arrow[r, "\tau_H^*"] & e_5 \arrow[r, phantom, "\equiv"] & e_4 \arrow[r, phantom, "\equiv"] & f_4 &
\end{tikzcd}
$$

The result follows. □

## 7 Conclusions

In this paper we have defined a compositional operational semantics for a core subset of CML, called $\mu$CML, and used it to develop at least the beginnings of an algebraic theory of CML programs based on an appropriate version of weak bisimulation equivalence. The operational semantics required an extension of the language to $\mu\text{CML}^+$ although it is worth pointing out that all of the added constructs can be defined in the core language $\mu$CML up to weak bisimulation equivalence:

Much research remains to be done. The algebraic theory of $\mu$CML, started in Section 5, needs to be developed to the extent that it can be used to characterise the semantic equivalence $\approx^n$. More generally both the operational semantics and the semantic equivalence should be extended to incorporate more of the features of CML. Of particular interest is the generation of new channel names. We believe that our operational semantics can be adapted to handle new channel generation but the semantic equivalence would need to be changed to an appropriate adaptation of *context bisimulation equivalence*, [31].

As pointed out in Section 3 our semantic equivalence, $\approx^n$, is based on the *late* version of bisimulations, [22]. This fits in quite well with the functional nature of CML but nevertheless it would be of interest to consider other variations. One can easily define an *early* version of $\approx^n$ or versions where silent moves are allowed to occur after a matching $\xrightarrow{l}$ move. However we have been unable to adapt Howe's method to show that these equivalences are preserved by $\mu$CML contexts.

In Section 3, we were forced to develop the theory of hereditary bisimulations because of the usual problems of $\tau$ actions resolving choice. In the sublanguage without always and **A**, we showed that weak bisimulation coincided with insensitive hereditary bisimulation, and so has a simpler and more elegant theory. This



theory has been investigated by the first author [8]. In this theory, it is possible to use CSP rather than CCS summation, and so weak bisimulation is respected by all contexts. As a side-effect of this, it is possible to remove the syntactic restriction that $[ge]$ can only be applied to guarded expressions. The third author has shown [19] that the resulting semantics can be presented in terms of Moggi's [23] monadic type system.

There has already been a considerable amount of research into the foundations of CML and related languages. Much of this is concerned with developing more detailed type systems, where types contain information on the behaviour of expressions as they evolve, [24]. Here we confine our remarks to work directly concerned with the development of semantic theories. We have already given a detailed comparison with the operational semantics given in [29, 30]. This semantics has been used in [5] to study an implementation of ML reference types using process generation. If we extend our approach to include channel generation then we could hope to give an algebraic treatment of their results. In [6, 7] there are a number of different semantics given to languages related to CML. A denotational semantics is given using the concept of "dynamic types" but it has not yet been related to any operationally based equivalence. An operational semantics is also given for a language called *FPI*. This contains many CML features but the author notes that accommodating any *spawn* or *fork* operator would be difficult. In [13, 3] the *spawn* operator is studied within the context of process algebras. The former gives a two-level operational semantics for a simple "pure" process algebra with *fork* and uses this to develop a semantic equivalence based on strong bisimulation; an axiomatisation is also given using an auxiliary operator called *forked*. The latter shows how the various algebraic theories of *ACP* can be adapted to support the addition of a *spawn* operator. This contains an lts based operational semantics for $ACP + spawn$ and their treatment of *spawn* has been used in [9] to give an operational semantics of a language which can be considered to be an untyped version of $\mu$CML. However bisimulation based equivalences are not developed in [9]; instead a testing equivalence is defined [14] and a fully-abstract denotational semantics based on Acceptance Trees is given.

Other languages which contain much in common with CML include *CHOCS* [32], *FACILE* [10], *PICT* [26], *ACTORS* [2] and *HO*$\pi$ [31]. Most of these are endowed with an operational semantics some of which are similar in spirit to ours. However we feel that our treatment of *spawn* and delayed computations is novel and hope that it can be used to good effect with other languages. Many of these languages also have associated with them bisimulation based semantic equivalences. Section 3 may be viewed as an extension of the research in [32] and the new equivalence $\approx^n$ can easily be adopted to languages such as *CHOCS* and *FACILE*. We have also already indicated that when we extend $\mu$CML to include channel generation it will be necessary to adopt the *context bisimulation equivalence*, originally developed in [31]. In short although semantic theories are being developed independently for these languages many of the techniques developed will find more general application.

## Appendix

This section is devoted to the proof of Proposition 6.6 and Proposition 6.7. But first we need some auxiliary results. The following three Propositions state elementary properties of the reduction contexts for $\mu$CML$^+$, introduced in Section 6 and we omit the proofs; they all use structural induction on contexts:

PROPOSITION A.1. *If $e \xrightarrow{\alpha} e'$ then $E^+[e] \xrightarrow{\alpha} E^+[e']$.*

PROPOSITION A.2. *If $E_1^+[e] \xrightarrow{l} f$ then either:*

- *$f = E_2^+[e]$ and for all $g$, $E_1^+[g] \xrightarrow{l} E_2^+[g]$, or*
- *$f = E_2^+[e']$, $e \xrightarrow{l'} e'$, and for all $g \xrightarrow{l'} g'$, $E_1^+[g] \xrightarrow{l} E_2^+[g']$.*

PROPOSITION A.3. *For any $E$ there is an $E^+$ such that for all $e$, $E[e] \xrightarrow{\tau_H}{}^* E^+[e]$.*

With these we can now prove Proposition 6.6:

PROPOSITION A.4. *$\equiv$ is a strong first-order bisimulation which respects housekeeping.*

PROOF. First observe that an alternative definition of $\equiv$ is as the smallest equivalence given by:

$$\frac{}{\Lambda \parallel e \equiv e} \qquad \frac{}{(e \parallel f) \parallel g \equiv e \parallel (f \parallel g)} \qquad \frac{}{e \parallel (f \parallel g) \equiv f \parallel (e \parallel g)}$$

$$\frac{}{(e \parallel f)\, g \equiv e \parallel (f\, g)} \qquad \frac{}{c\,(e \parallel f) \equiv e \parallel (c\, f)} \qquad \frac{}{(e \parallel f, g) \equiv e \parallel (f, g)}$$

$$\frac{}{\mathsf{let}\, x = e \parallel f\, \mathsf{in}\, g \equiv e \parallel \mathsf{let}\, x = f\, \mathsf{in}\, g} \qquad \frac{}{\mathsf{if}\, e \parallel f\, \mathsf{then}\, g\, \mathsf{else}\, h \equiv e \parallel \mathsf{if}\, f\, \mathsf{then}\, g\, \mathsf{else}\, h}$$

$$\frac{e \equiv f}{E[e] \equiv E[f]}$$

Then show by induction on the proof of this alternative that $\equiv$ satisfies the required properties to be a first-order strong bisimulation which preserves housekeeping. □

The next result shows that the auxilarly predicates used in the reduction semantics of $\mu$CML$^{cv}$, $\Longrightarrow$, have their exact counterparts in the $\mu$CML$^+$ semantics:

PROPOSITION A.5.

1. *$ge \xrightarrow{k!v} e$ iff $ge \overset{k}{\bowtie} k?$ **with** $(e, v)$,*
2. *$ge \xrightarrow{k?x} e$ iff $ge \overset{k}{\bowtie} k!x$ **with** $(e, ())$,*
3. *$ge \xrightarrow{\tau} e$ iff $ge \rhd e$, and*
4. *if $ge_1 \overset{k}{\bowtie} ge_2$ **with** $(e_1, e_2)$ then $ge_i \xrightarrow{k!v} e_i$ and $ge_j \xrightarrow{k?v} e_j$.*

PROOF. A routine structural induction. □

We these results we can now give the proof of Proposition 6.7, which for convenience we restate:



PROPOSITION A.6. *If $C \xrightarrow{\tau_H}{}^* e$ and $e$ is tidy, then the following diagrams can be completed:*

$$\begin{tikzcd} C \arrow[r, "\tau_H^*"] \arrow[d, mapsto, "l"'] & e \\ C' & \end{tikzcd} \quad \textit{as} \quad \begin{tikzcd} C \arrow[r, "\tau_H^*"] \arrow[d, mapsto, "l"'] & e \arrow[d, "l"'] \\ C' \arrow[r, "\tau_H^{*\circ}"] & \equiv e' \end{tikzcd}$$

*and:*

$$\begin{tikzcd} C \arrow[r, "\tau_H^*"] & e \arrow[d, "l"'] \\ & e' \end{tikzcd} \quad \textit{as} \quad \begin{tikzcd} C \arrow[r, "\tau_H^*"] \arrow[d, mapsto, "l"'] & e \arrow[d, "l"'] \\ C' \arrow[r, "\tau_H^{*\circ}"] & \equiv e' \end{tikzcd}$$

PROOF. The first diagram is completed by case analysis of $C \overset{l}{\longmapsto} C'$. We shall prove some of the cases, as the others are similar.

- If $C \overset{\tau}{\longmapsto} C'$ from the <u>const</u> rule, then $C = E_1^+[E_2[c\,v]]$ and $C' = E_1^+[E_2[c\,v]$. Then by Propositions A.1 and A.3:

$$\begin{tikzcd} C \arrow[r, equal] \arrow[d, mapsto, "\tau"'] & E_1^+[E_2[c\,v]] \arrow[r, "\tau_H^*"] \arrow[d, mapsto, "\tau"'] & E_1^+[E_2^+[c\,v]] \arrow[r, equal] & e \arrow[d, "\tau"'] \\ C' \arrow[r, equal] & E_1^+[E_2[\delta(c, v)]] \arrow[r, "\tau_H^*"] & E_1^+[E_2^+[\delta(c\,v)]] \arrow[r, phantom, "\equiv"] & E_1^+[E_2^+[\Lambda \parallel \delta(c\,v)]] \end{tikzcd}$$

- If $C \overset{\surd v}{\longmapsto} C'$ then $C = C'' \parallel v$ and $C' = C'' \parallel \Lambda$, so:

$$\begin{tikzcd} C \arrow[r, equal] \arrow[d, mapsto, "\surd v"'] & C'' \parallel v \arrow[r, "\tau_H^*"] \arrow[d, mapsto, "\surd v"'] & e'' \parallel v \arrow[r, equal] & e \arrow[d, "\surd v"'] \\ C' \arrow[r, equal] & C'' \parallel \Lambda \arrow[r, "\tau_H^*"] & e'' \parallel \Lambda \arrow[r, phantom, "\equiv"] & e'' \parallel \Lambda \end{tikzcd}$$

- If $C \overset{k!v}{\longmapsto} C'$ then (from the definition of $C \overset{k!v}{\longmapsto} C'$ and the <u>comm</u> rule) $C = E_1^+[E_2[\mathsf{sync}[ge]]]$, $C = E_1^+[E_2[e]]$, and $ge \overset{k}{\bowtie} k?$ **with** $(e, v)$, so by Proposition A.5, $ge \xrightarrow{k!v} e$, and so by Propositions A.1 and A.3:

$$\begin{tikzcd} C \arrow[r, equal] \arrow[d, mapsto, "k!v"'] & E_1^+[E_2[\mathsf{sync}[ge]]] \arrow[r, "\tau_H^*"] \arrow[d, mapsto, "k!v"'] & E_1^+[E_2^+[ge]] \arrow[r, equal] & e \arrow[d, "k!v"'] \\ C' \arrow[r, equal] & E_1^+[E_2[e]] \arrow[r, "\tau_H^*"] & E_1^+[E_2^+[e]] \arrow[r, phantom, "\equiv"] & E_1^+[E_2^+[e]] \end{tikzcd}$$

The second diagram is completed by induction on $C$. We shall prove some of the cases, as the others are similar.

If $C = E[f]$, $E$ is a one-level deep reduction context for both $\mu$CML$^+$ and $\mu$CML$^{cv}$, $e = E[g]$, $f \xrightarrow{\tau_H}{}^* g$, $e' = E[g']$ and $g \xrightarrow{\alpha} g'$ then by induction $f \overset{l}{\longmapsto} C' \xrightarrow{\tau_H}{}^* f' \equiv g'$ and we can show by induction on $E$ that $E[g] \xmapsto{l} \equiv E[C]$ so by Propositions 6.6:

$$\begin{tikzcd}
C \arrow[r, equal] \arrow[d, "\alpha"'] & E[f] \arrow[r, "\tau_H^*"] & E[g] \arrow[r, equal] \arrow[dd, "\alpha"'] & e \arrow[dd, "\alpha"'] \\
C'' \arrow[r, equal] \arrow[dr, "\tau_H^{*\circ}"] & E[f'] \arrow[dr, "\tau_H^{*\circ}"] & & \\
& f'' \arrow[r, equal] & E[g'] \arrow[r, equal] & e'
\end{tikzcd}$$

Otherwise:

- If $C = c\,f$ then $f \xrightarrow{\tau_H}{}^{*} g$, $g$ is tidy and $c\,g \xrightarrow{\tau_H}{}^{*} e$, so either:
  - $c = \mathsf{sync}$, $e = g' \parallel ge$, $g \xrightarrow{\surd[ge]} g'$, and $f \xrightarrow{\tau_H}{}^{*} g$, so by induction and the definition of $\xmapsto{\surd v}$, $f = g = [ge]$ and $g' = \Lambda$, so $e' = \Lambda \parallel g''$ and $ge \xrightarrow{\alpha} g''$, so by Proposition A.5, $\mathsf{sync}[ge] \xmapsto{\alpha} g''$, and so:

$$\begin{tikzcd}
C \arrow[r, equal] \arrow[d, "\alpha"'] & \mathsf{sync}[ge] \arrow[r, "\tau_H^*"] & \Lambda \parallel ge \arrow[r, equal] \arrow[d, "\alpha"'] & e \arrow[d, "\alpha"'] \\
g'' \arrow[r, "\tau_H^{*\circ}"] & g'' \arrow[r, phantom, "\equiv"] & \Lambda \parallel g'' \arrow[r, equal] & e'
\end{tikzcd}$$

  - $c = \mathsf{spawn}$, $e = \mathsf{spawn}\,g$, $e' = g' \parallel v() \parallel ()$ and $g \xrightarrow{\surd v} g'$, so by induction and the definition of $\xmapsto{\surd v}$, $f = g = v$ and $g' = \Lambda$, and so:

$$\begin{tikzcd}
C \arrow[r, equal] \arrow[d, "\tau"'] & \mathsf{spawn}\,v \arrow[r, "\tau_H^*"] & \mathsf{spawn}\,g \arrow[r, equal] \arrow[d, "\tau"'] & e \arrow[d, "\tau"'] \\
v() \parallel () \arrow[r, "\tau_H^{*\circ}"] & v() \parallel () \arrow[r, phantom, "\equiv"] & \Lambda \parallel v() \parallel () \arrow[r, equal] & e'
\end{tikzcd}$$

  - or $e' = g' \parallel \delta(c,v)$ and $g \xrightarrow{\surd v} g'$, so by induction and the definition of $\xmapsto{\surd v}$, $f = g = v$ and $g' = \Lambda$, and so:

$$\begin{tikzcd}
C \arrow[r, equal] \arrow[d, "\tau"'] & c\,v \arrow[r, "\tau_H^*"] & \mathsf{spawn}\,g \arrow[r, equal] \arrow[d, "\tau"'] & e \arrow[d, "\tau"'] \\
\delta(c,v) \arrow[r, "\tau_H^{*\circ}"] & \delta(c,v) \arrow[r, phantom, "\equiv"] & \Lambda \parallel \delta(c,v) \arrow[r, equal] & e'
\end{tikzcd}$$

- If $C = f_1\,f_2$ then $f_1 \xrightarrow{\tau_H}{}^{*} g_1 \xrightarrow{\surd v} g_1'$ where $v = \mathsf{fix}(x = \mathsf{fn}\ y \Rightarrow g_3)$, $f_2 \xrightarrow{\tau_H}{}^{*} g_2$, $e = g_1' \parallel \mathsf{let}\,y = g_2\,\mathsf{in}\,g_3[v/x]$, so by induction and the definition of $\xmapsto{\surd v}$, $f_1 = g_1 = v$ and $g_1' = \Lambda$, and so either:
  - $e' = g_1' \parallel \mathsf{let}\,y = g_2'\,\mathsf{in}\,g_3[v/x]$ and $g_2 \xrightarrow{\alpha} g_2'$ so by induction (up to associativity of $\parallel$ and $\Lambda$ being a left unit), $f_2 \xmapsto{\alpha} C' \parallel f_2' \xrightarrow{\tau_H}{}^{*\circ} f_3 \parallel f_2'' \equiv g_2'$, and so:

$$\begin{tikzcd}
C \arrow[r, equal] \arrow[d, "\alpha"'] & v\,f_2 \arrow[r, "\tau_H^*"] & \Lambda \parallel \mathsf{let}\,y = g_2\,\mathsf{in}\,g_3[v/x] \arrow[r, equal] \arrow[d, "\alpha"'] & e \arrow[d, "\alpha"'] \\
C' \parallel v\,f_2' \arrow[r, "\tau_H^{*\circ}"] & f_3 \parallel \mathsf{let}\,y = f_2''\,\mathsf{in}\,g_3[v/x] \arrow[r, phantom, "\equiv"] & \Lambda \parallel \mathsf{let}\,y = g_2'\,\mathsf{in}\,g_3[v/x] \arrow[r, equal] & e'
\end{tikzcd}$$

  - or $e' = g_1' \parallel g_2' \parallel g_3[v/x][w/y]$ and $g_2 \xrightarrow{\surd w} g_2'$, so by induction and the definition of $\xmapsto{\surd v}$, $f_2 = g_2 = w$ and $g_2' = \Lambda$, and so:

$$\begin{tikzcd}
C \arrow[r, equal] \arrow[d, "\tau"'] & v\,w \arrow[r, "\tau_H^*"] & \Lambda \parallel \mathsf{let}\,y = w\,\mathsf{in}\,g_3[v/x] \arrow[r, equal] \arrow[d, "\tau"'] & e \arrow[d, "\tau"'] \\
g_3[v/x][w/y] \arrow[r, "\tau_H^*"] & g_3[v/x][w/y] \arrow[r, phantom, "\equiv"] & \Lambda \parallel \Lambda \parallel g_3[v/x][w/y] \arrow[r, equal] & e'
\end{tikzcd}$$

The result follows. □



## References


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

[2] G. Agha, I. Mason, S. Smith, and C. Talcott. A foundation for actor computation. *J. Functional Programming*, 1994.

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

[4] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoret. Comput. Sci.*, 37:77–121, 1985.

[5] Dave Berry, Robin Milner, and David N. Turner. A semantics for ML concurrency primitives. In *Proc. POPL 92*, 1992.

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

[7] M. Debabi. *Integration de Paradigmes de Programmation Paralle, Fonctionnelle et Imperative*. Ph.D thesis, Universite D'Orsay, 1994.

[8] William Ferreira. *Semantic Theories for Concurrent ML*. D.Phil thesis, COGS, Sussex Univ., 1995. In preparation.

[9] William Ferreira and Matthew Hennessy. Towards a semantic theory of CML. Technical report 95:02, COGS, Sussex Univ., 1995.

[10] A. Giacalone, P. Mishra, and S. Prasad. Facile: A symmetric integration of concurrent and functional programming. In *Proc. Tapsoft 89*, volume 352 of *LNCS*, pages 184–209. Springer-Verlag, 1989.

[11] Andrew Gordon. Bisimilarity as a theory of functional programming. In *Proc. MFPS 95*, number 1 in Electronic Notes in Comp. Sci. Springer-Verlag, 1995.

[12] Carl Gunter. *Semantics of Programming Languages*. MIT Press, 1992.

[13] K. Havelund. *The Fork Calculus: Towards a Logic for Concurrent ML*. Ph.D thesis, École Normale Superieur, Paris, 1994.

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

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

[16] Sören Holmström. PFL: A functional language for parallel programming. In *Proc. Declarative Programming Workshop*, pages 114–139, 1983.

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

[18] Douglas Howe. Proving congruence of simulation orderings in functional languages. Unpublished manuscript, 1992.

[19] Alan Jeffrey. A fully abstract semantics for a concurrent functional language with monadic types. In *Proc. LICS 95*, pages 255–264, 1995.

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

[21] Robin Milner. The polyadic π-calculus: a tutorial. In *Proc. International Summer School on Logic and Algebra of Specification*, Marktoberdorf, 1991.

[22] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile proceses. *Inform. and Comput.*, 100(1):1–77, 1992.

[23] Eugenio Moggi. Notions of computation and monad. *Inform. and Comput.*, 93:55–92, 1991.

[24] F. Nielson and H. R. Nielson. From CML to process algebras. Report DAIMI FN-19, Dept. Comp. Sci., Aarhus University, 1993.

[25] Sivaswami Nikhil. *Id Reference Manual*. MIT Lab. for Comp. Sci., 1990.

[26] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. Technical report in preparation; available electronically from http://www.cl.cam.ac.uk/users/bcp1000/ftp/index.html, 1995.

[27] Gordon Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoret. Comput. Sci.*, 1:125–159, 1975.

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

[29] John Reppy. An operational semantics of first-class synchronous operations. Technical report TR 91-1232, Dept. Comp. Sci., Cornell Univ., 1991.

[30] John Reppy. *Higher-Order Concurrency*. Ph.D thesis, Cornell Univ., 1992.

[31] Davide Sangiorgi. *Expressing Mobility in Process Algebras: First-order and Higher-order Paradigms*. Ph.D thesis, LFCS, Edinburgh Univ., 1992.

[32] Bent Thomsen. A theory of higher order communicating systems. *Inform. and Comput.*, 116(1):38–57, 1995.

[33] A. Wright and M. Felleisen. A syntactic approach to type soundness. Technical report TR91-160, Dept. of Comp. Sci., Rice Univ., 1991.