A Theory of Weak Bisimulation for Core CML

WILLIAM FERREIRA†

Computing Laboratory
University of Cambridge

MATTHEW HENNESSY AND ALAN JEFFREY‡

School of Cognitive and Computing Sciences
University of Sussex

Abstract

Concurrent ML (CML) is an extension of Standard ML of New Jersey with concurrent features similar to those of process algebra. In this paper, we build upon John Reppy's reduction semantics for CML by constructing a compositional operational semantics for a fragment of CML, based on higher-order process algebra. Using the operational semantics we generalise the notion of weak bisimulation equivalence to build a semantic theory of 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, (Giacalone *et al.*, 1989; Holmström, 1983; Nikhil, 1990). Concurrent ML (CML) (Reppy, 1991a; Reppy, 1992; Panangaden & Reppy, 1996) 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, and a new function for spawning concurrent threads of computation. Thus the language has all the functional and higher-order features of ML, but in addition programs also have the ability to communicate with each other by transmitting values along communication channels.

In (Reppy, 1992), a reduction style operational semantics is given for a subset of CML called λ_{cv}, which may be viewed as a concurrent version of the call-by-value λ-calculus of (Plotkin, 1975). Reppy's semantics gives reduction rules for whole programs, not for program fragments. It is not *compositional*, in that the semantics of a program is not defined in terms of the semantics of its subterms. Reppy's semantics is designed to prove properties about programs (for example type safety), and not about program fragments (for example equational reasoning).

In this paper we construct a compositional operational semantics in terms of a labelled

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

transition system, for a core subset of CML which we call μCML. This semantics not only describes the evaluation steps of programs, as in (Reppy, 1992), but also their communication potentials in terms of their ability to input and output values along communication channels. This semantics extends the semantics of higher-order processes (Thomsen, 1995) with types and first-class functions.

We then proceed to demonstrate the usefulness of this semantics by using it to define a version of *weak bisimulation*, (Milner, 1989), suitable for μCML. We prove that, modulo the usual problems associated with the choice operator of CCS, our chosen equivalence is preserved by all μCML 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 λ-calculus are given and we also show that certain algebraic laws common to process algebras, (Milner, 1989), hold.

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

In Section 2 we describe μCML, a monomorphically typed core subset of CML, which nonetheless includes base types for channel names, booleans and integers, and type constructors for pairs, functions, and delayed computations which are known as *events*. μCML also includes a selection of the constructs and constants for manipulating event types, such as `transmit` and `receive` for constructing basic events for sending and receiving values, `wrap` for combining delayed computations, `choose` for selecting between delayed computations, and a function `spawn` for launching new concurrent threads of computation within a program. The major omission is that μCML has no facility for generating new channel names. However we believe that this can be remedied by using techniques common to the π-calculus, (Milner, 1991; Milner *et al.*, 1992; Sangiorgi, 1992).

In the remainder of this section we present the operational semantics of μCML in terms of a labelled transition system. In order to describe all possible states which can arise during the computation of a well-typed μCML 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 (Reppy, 1992) to define λ_{cv}, i.e. constants to denote certain delayed computations. This extended language, which we call $\mu\mathrm{CML}^{cv}$, essentially coincides with the λ_{cv}, the language used in (Reppy, 1992), except for the omissions cited above. However to obtain a compositional semantics we make further extensions to $\mu\mathrm{CML}^{cv}$. We add a parallel operator $\parallel$, commonly used in process algebras, which allows us to use programs in place of the multisets of programs of (Reppy, 1992).

The final addition is more subtle; we include in $\mu\mathrm{CML}^{cv}$ expressions which correspond to the `sync`ed versions of Reppy's constants for representing delayed computations. Thus the labelled transition system uses as states programs from a language which we call $\mu\mathrm{CML}^{+}$. This language is a superset of $\mu\mathrm{CML}^{cv}$, which is our version of Reppy's λ_{cv}, which in turn is a superset of μCML, our mini-version of CML. The following diagram

indicates the relationships between these languages:

$$\begin{tikzcd} \mu\mathrm{CML} \arrow[r] \arrow[d] & \mu\mathrm{CML}^{cv} \arrow[r] \arrow[d] & \lambda_{cv} \\ \mathrm{CML} & \mu\mathrm{CML}^{+} & \end{tikzcd}$$

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, (Milner, 1989), and then consider adaptations of *higher-order* and *irreflexive* bisimulations from (Thomsen, 1995). 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 μCML contexts. This is an application of the proof method originally suggested in (Howe, 1989) 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 μCML 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 λ-calculus (Plotkin, 1975) and process algebras (Milner, 1989).

In Section 6 we show that, up to weak bisimulation equivalence, our semantics coincides with the reduction semantics for λ_{cv} presented in (Reppy, 1992). This technical result applies only to the common sub-language, namely $\mu\mathrm{CML}^{cv}$.

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 μCML, a subset of Concurrent ML (Reppy, 1991a; Reppy, 1992; Panangaden & Reppy, 1996). We describe the syntax, including a typing system, and an operational semantics in terms of a labelled transition system.

Unfortunately, there is not enough space in this paper to provide an introduction to programming in CML: see (Panangaden & Reppy, 1996) for a discussion of the design and philosophy of CML.

The type expressions for our language are given by:

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

Thus we have three base types, `unit`, `bool` and `int`; the latter two are simply examples of useful base types and one could easily include more. These types are closed under four constructors: pairing, function space, and the less common `chan` and `event` type constructors.

Our language may be viewed as a typed λ-calculus augmented with the type constructors $A\,\mathsf{chan}$ for *communication channels* sending and receiving data of type A, and $A\,\mathsf{event}$ for constructing *delayed computations* of type A.

Let $Chan_A$ be a type-indexed family of disjoint sets of channel names, ranged over by k, and let Var denote a set of variables ranged over by x, y and z. The expressions of μCML are given by the following abstract syntax:

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

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, let-binding and branching constructors, and two forms of application: the application of one expression to another, $e\,e$, the application of a constant to an expression, $c\,e$.

There is also a syntactic category of *value* expressions Val, used in giving a semantics to call-by-value functions and communicate-by-value channels. They are restricted in form: either a variable, a recursively defined function, $\mathsf{fix}(x = \mathsf{fn}\,y \Rightarrow e)$, or a predefined literal value for the base types. We will use some syntax sugar, writing $\mathsf{fn}\,y \Rightarrow e$ for $\mathsf{fix}(x = \mathsf{fn}\,y \Rightarrow e)$ when x does not occur in e, and $e; f$ for $\mathsf{let}\,x = e\,\mathsf{in}\,f$ when x does not occur in f.

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 fst and snd, together with the set of constants for manipulating *delayed computations* taken directly from (Reppy, 1992):

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

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. (Since we are modelling a

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

Figure 1a. Type rules for μCML constant functions

$$
\frac{}{\Gamma, x : A \vdash x : A} \qquad \frac{\Gamma \vdash y : B}{\Gamma, x : A \vdash y : B}[x \neq y]
$$

$$
\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool}} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool}} \qquad \frac{}{\Gamma \vdash k : A\,\mathsf{chan}}[k \in Chan_A]
$$

$$
\frac{}{\Gamma \vdash () : \mathsf{unit}} \qquad \frac{}{\Gamma \vdash n : \mathsf{int}} \qquad \frac{\Gamma, x : A \to B, y : A \vdash e : B}{\Gamma \vdash \mathsf{fix}(x = \mathsf{fn}\, y \Rightarrow e) : A \to B}
$$

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

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

Figure 1b. Type rules for μCML expressions.

call-by-value language, we have limited substitution to values $e[v/x]$ rather than the more general $e[f/x]$. In order to model alpha-conversion, we have therefore included variables as possible values.)

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 (Reppy, 1992) for λ_{cv}. Note that many of the constants (such as $\mathsf{choose} : A\,\mathsf{event} * A\,\mathsf{event} \to A\,\mathsf{event}$) have a family of types.

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* Γ, 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 judgement should be read as "in the type assignment Γ 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, (Gunter, 1992).

We now turn our attention to the operational semantics. In (Reppy, 1992; Berry *et al.*, 1992) a reduction semantics is given to λ_{cv} and since $\mu\mathrm{CML}^{cv}$ is a subset of λ_{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 run-time environment necessary for its evaluation, and τ is Milner's notation for a silent action. 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,
- $e \xrightarrow{\surd v} e'$, representing the production of the value v, with a side effect e',
- $e \xrightarrow{k?x} e'$, 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 are summarized in Figure 3.

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 μCML expressions can spawn subprocesses before returning a value, so we have to allow expressions to continue evaluation even after they have returned a result. For example in the expression:

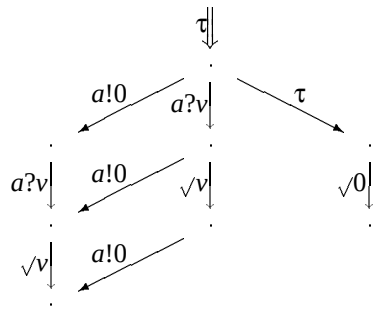
$$\mathsf{spawn}(\mathsf{fn}\ () \Rightarrow \mathsf{sync}\,(\mathsf{transmit}(0, a))); \mathsf{sync}(\mathsf{receive}\ a)$$

one possible reduction is (where $\stackrel{\tau}{\Longrightarrow}$ indicates a sequence of τ-reductions):

$$\mathsf{spawn}(\mathsf{fn}\ () \Rightarrow \mathsf{sync}\,(\mathsf{transmit}(0, a))); \mathsf{sync}(\mathsf{receive}\ a) \stackrel{\tau}{\Longrightarrow} \xrightarrow{a?1} \xrightarrow{\surd 1} \xrightarrow{a!0}$$

where the process returns the value 1 before outputting 0. For this reason we need a reduction $e \xrightarrow{\surd v} e'$ rather than the more usual termination $e \downarrow v$. The following diagram illustrates all of the possible transitions from this expression:

$$\mathsf{spawn}(\mathsf{fn}\ () \Rightarrow \mathsf{sync}\,(\mathsf{transmit}(0, a))); \mathsf{sync}(\mathsf{receive}\ a)$$



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 (Berry *et al.*, 1992; Reppy, 1992), 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, (Bergstra & Klop, 1985; Milner, 1989), 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 a parallel operator, $e \Vert f$. This has the same operational rules as in CCS, allowing reduction of both processes:

$$\frac{e \xrightarrow{\alpha} e'}{e \Vert f \xrightarrow{\alpha} e' \Vert f} \qquad \frac{f \xrightarrow{\alpha} f'}{e \Vert f \xrightarrow{\alpha} e \Vert f'}$$

and communication between the processes:

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

The assymetry is introduced by termination (a feature missing from CCS). A CML process has a *main thread of control*, and only the main thread can return a value. By convention, we write the main thread on the right, so the rule is:

$$\frac{f \xrightarrow{\surd v} f'}{e \Vert f \xrightarrow{\surd v} e \Vert f'}$$

There is no corresponding symmetric rule. For example:

$$() \Vert 1 \xrightarrow{\surd 1} () \Vert \Lambda \qquad () \Vert 1 \not\xrightarrow{\surd ()} \Lambda \Vert 1$$

Since the only difference between concurrent processes is which term can return a value, concurrency is associative and symmetric on the left, so $e \Vert f \Vert g$ is bisimilar to $f \Vert e \Vert g$. In general, we can regard $n+1$ processes in parallel:

$$e_1 \Vert \cdots \Vert e_n \Vert f$$

as being a multiset of spawned threads $e_1, \cdots, e_n$ plus one main thread of control f, corresponding to the use of multi-sets in the reduction semantics of (Berry *et al.*, 1992; Reppy, 1992).

Concurrent processes are generated using the constant application $\mathsf{spawn}\, e$. A first attempt to write the semantics for $\mathsf{spawn}\, e$ would be the rule:

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

One step in the evaluation of $\mathsf{spawn}(\mathsf{fn}\ y \Rightarrow e)$ leads to two expressions running in parallel, one being the spawned function application $(\mathsf{fn}\ y \Rightarrow e)\,()$ and the other the default value $()$ which results from every application of spawn. However, this rule for $\mathsf{spawn}\, e$ is not general enough. Firstly, it ignores the fact that the expression e may need to perform some computation before returning a function, which is captured by instantiating the static rule for constant application as:

$$\frac{e \xrightarrow{\alpha} e'}{\mathsf{spawn}\, e \xrightarrow{\alpha} \mathsf{spawn}\, e'}$$

Secondly, e may have spawned some concurrent processes before returning a function, and these should carry on evaluation, so we use the silent rule for constant application:

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

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 ef, first e is evaluated to a value of functional form and then the evaluation of f is initiated. This is represented by the rules:

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

(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 ef, we now evaluate f to a value which is then substituted into g for y. This is represented by the two rules:

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

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

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

Here, borrowing the notation of (Reppy, 1992), we use the function δ 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 spawn, which result from the fact that $\delta(\mathsf{spawn}, v) = v() \Vdash ()$. The definition of δ for all constants in the language is given in Figure 2f. 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 $\Vdash$ into the language we can treat all constants uniformly, unlike (Reppy, 1992) where spawn and 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:
$$\frac{e \xrightarrow{\surd v} e'}{(e, f) \xrightarrow{\tau} e' \Vdash \mathsf{let}\,x = f\,\mathsf{in}\,\langle v, x\rangle}$$

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

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

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

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

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

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

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

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

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

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

Figure 2c. Operational semantics: silent rules

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

Figure 2d. Operational semantics: axioms

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

Figure 2e. Operational semantics: grammar of labels

$$\begin{array}{rcl@{\qquad}rcl}
\delta(\mathsf{fst},\langle v,w\rangle) & = & v & \delta(\mathsf{snd},\langle v,w\rangle) & = & w \\
\delta(\mathsf{add},\langle m,n\rangle) & = & m+n & \delta(\mathsf{mul},\langle m,n\rangle) & = & m \times n \\
\delta(\mathsf{leq},\langle m,n\rangle) & = & m \leq n & & & \\
 & & & & & \\
\delta(\mathsf{transmit},\langle k,v\rangle) & = & [k!v] & \delta(\mathsf{receive},k) & = & [k?] \\
\delta(\mathsf{choose},\langle [ge_1],[ge_2]\rangle) & = & [ge_1 \oplus ge_2] & \delta(\mathsf{wrap},\langle [ge],v\rangle) & = & [ge \Rightarrow v] \\
\delta(\mathsf{never},()) & = & [\Lambda] & \delta(\mathsf{always},v) & = & [\mathbf{A}v] \\
\delta(\mathsf{spawn},v) & = & v() \Vdash () & \delta(\mathsf{sync},[ge]) & = & ge
\end{array}$$

Figure 2f. Operational semantics:reduction of constants

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

Figure 3a. Syntax of μCML

$$
\begin{array}{rcl}
v, w \in Val & ::= & \cdots \mid \langle v, v \rangle \mid [ge] \\
ge \in GExp & ::= & v!v \mid v? \mid ge \Rightarrow v \mid ge \oplus ge \mid \Lambda \mid \mathbf{A}v
\end{array}
$$

Figure 3b. Syntax of $\mu\mathrm{CML}^{cv}$

$$
e, f, g \in Exp \quad ::= \quad \cdots \mid ge \mid e \Vdash e
$$

Figure 3c. Syntax of $\mu\mathrm{CML}^{+}$

constant sync, of type $A\,\mathsf{event} \to A$. Thus for example the expression $\mathsf{receive}\, k$ 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 $\mathsf{sync}(\mathsf{receive}\, k)$ 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 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:

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

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 event from the basic language μCML. The constant $\mathsf{receive}$ is of type $A\,\mathsf{chan} \to A\,\mathsf{event}$ and therefore

the evaluation of the expression $\mathsf{receive}\, e$ proceeds by first evaluating e to a value of type $A\,\mathsf{chan}$ 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 $k?$ be a guarded expression for any k and A, with the associated rule:

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

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

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

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

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

Intuitively this may be read as $k!v$ 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 semantics we consider for input is the *late* semantics, where the reduction rule binds a new variable x:

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

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

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

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

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

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

The constant wrap is of type $A\,\mathsf{event} * (A \rightarrow B) \rightarrow B\,\mathsf{event}$. 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 $A \rightarrow B$. 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:

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

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 always construct, of type $A \rightarrow A\,\mathsf{event}$, evaluates its argument to a value v, and then

returns a trivial a 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 always is then captured by the rule:

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

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 choose has the type $A\mathsf{event} * A\mathsf{event} \rightarrow A\mathsf{event}$. 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:

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

The notation $\oplus$, introduced in (Reppy, 1992), is unfortunate, as it is used in (Hennessy, 1988) 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 μCML. We now summarise, giving the precise definitions of the new syntax. For the purposes of comparison with the reduction semantics of λ_{cv}, (Reppy, 1992), it is convenient to view the extension to μCML in two stages. The first is obtained by adding the new syntactic category of guarded expressions, and two new constructors for values:

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

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

$$e \in Exp \quad ::= \quad \cdots \mid ge \mid e \mathbin{+\!\!\!+} e$$

and type judgements for all the extra constructs appear in Figure 4.

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*:

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

which are typed with judgements $\vdash l : A$ in Figure 5, 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.

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

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

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

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

Fig. 4. Type rules for extra μCML$^+$ constructs

$$\frac{}{\Gamma \vdash \tau : A} \qquad \frac{\Gamma \vdash v : A}{\Gamma \vdash \surd v : A} \qquad \frac{}{\Gamma \vdash k?x : A} \qquad \frac{\Gamma \vdash w : B}{\Gamma \vdash k!w : A}[k \in Chan_B]$$

Fig. 5. Type rules for labels

It is worth pointing out that the context rules are asymmetric for the propagation of value production though the context $\Vdash$; in $e \Vdash f$ only the computation thread f can produce a value. This is in agreement with the reduction semantics of (Reppy, 1992) 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 $\vdash e : A$ in μCML$^+$

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

Proof
By rule induction on the inferences. □

3 Weak Bisimulation Equivalence

In this section we demonstrate the usefulness of our operational semantics by providing μCML$^+$ 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 μCML$^+$.

We first show how to adapt the notion of strong bisimulation equivalence to μCML$^+$. 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 $e \xrightarrow{k?x} f$ 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 $e\ \mathcal{R}_{\Gamma,A}\ f$ then $\Gamma \vdash e : A$ and $\Gamma \vdash f : A$. We will often elide the subscripts from relations, for example writing $e\ \mathcal{R}\ f$ for $e\ \mathcal{R}_{\Gamma,A}\ f$ when context makes the type obvious. Let a *closed type-indexed* relation $\mathcal{R}$ be an open type-indexed relation where Γ 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:

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

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

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

With this notation we can now define strong bisimulations over μCML$^+$ 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:

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

Note the use of the open extension $\mathcal{R}^\circ$. This means, for example, that if $e_1\ \mathcal{R}\ e_2$ we require that the move $e_1 \xrightarrow{k?x} f_1$ be matched by a move $e_2 \xrightarrow{k?x} f_2$ where f_2 is such that for all values v of the appropriate type $f_1[v/x]\ \mathcal{R}\ f_2[v/x]$. Thus in the terminology of (Milner *et al.*, 1992) our definition corresponds to the *late* version of bisimulation. (An alternative would be *early* bisimulation where input moves are labelled with closed values rather than variables. This is computationally more appealling, but it is an open problem whether the techniques of the next section can be applied to open 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 the identity relation I and the relation composition $\mathcal{R}\mathcal{R}$. The result follows. $\square$

Unfortunately, $\sim^1$ is not a congruence for μCML$^+$, 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 μCML contexts:

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

since the left hand side can perform the move:

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

but this can only be matched by the right hand side up to strong bisimulation:

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

In fact, it is easy to verify that the only first-order strong bisimulation which is a congruence for μCML is the identity relation.

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

$$\frac{}{\tau\, \mathcal{R}^l_A\, \tau} \qquad \frac{v\, \mathcal{R}_A\, w}{\surd v\, \mathcal{R}^l_A\, \surd w} \qquad \frac{}{k?x\, \mathcal{R}^l_A\, k?x} \qquad \frac{v\, \mathcal{R}_B\, w}{k!v\, \mathcal{R}^l_A\, k!w}[k \in Chan_B]$$

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

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

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 $(\mathsf{fn}\, x \Rightarrow x)\, 0 = 0$. We require the coarser *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 $\stackrel{\tau}{\longrightarrow}$, and let $\stackrel{l}{\Longrightarrow}$ be $\stackrel{\varepsilon}{\Longrightarrow}\stackrel{l}{\longrightarrow}$ (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:

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

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

Proposition 3.3
$\approx^1$ is an equivalence.

Proof
Similar to the proof of Proposition 3.1. $\square$

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

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

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. $\square$

However, $\approx^h$ is still not a congruence, for the usual reason that weak bisimulation equivalence $\approx$ is not a congruence for CCS summation. Recall from (Milner, 1989) that in CCS $\mathbf{0} \approx \tau.\mathbf{0}$ but $a.\mathbf{0}+\mathbf{0} \not\approx a.\mathbf{0}+\tau.\mathbf{0}$. We can duplicate this counter-example in μCML$^+$ since the CCS operator $+$ corresponds to the μCML$^+$ operator $\oplus$ and $\mathbf{0}$ corresponds to Λ. However $\oplus$ may only be applied to *guarded expressions* and therefore we need a *guarded expression* which behaves like $\tau.\mathbf{0}$; the required expression is $\mathbf{A}[\Lambda] \Rightarrow \mathsf{sync}$. Thus:

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

since the right hand side has only one reduction:

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

but:

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

because the only reduction of $\Lambda \oplus k!0$ is $\Lambda \oplus k!0 \xrightarrow{k!0} \Lambda \oplus \Lambda$ and:

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

This counter-example can also be replicated using the restricted syntax of μCML. We have:

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

since the left hand side has only one reduction:

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

and the right hand side 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 left hand side has only one reduction:

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

whereas the right hand side has the reduction:

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

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

$$\begin{array}{ccc} e_1 & =^h & e_2 \\ {\scriptstyle l_1}\downarrow & & \\ e_1' & & \end{array} \quad \text{as} \quad \begin{array}{ccc} e_1 & =^h & e_2 \\ {\scriptstyle l_1}\downarrow & & {\scriptstyle l_2}\Downarrow \\ e_1' & \approx^h & e_2' \end{array} \text{ where } l_1 \approx^{h^l} l_2$$

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 μCML counter-example for $\approx^h$ being a congruence also applies to $=^h$; i.e.

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

but:

$$\begin{aligned}&\mathsf{sync}(\mathsf{choose}(\mathsf{never}(), \mathsf{transmit}(k,0)))\\ &\quad \neq^h \mathsf{sync}(\mathsf{choose}(\mathsf{wrap}(\mathsf{always}(\mathsf{never}()), \mathsf{sync}), \mathsf{transmit}(k,0)))\end{aligned}$$

This failure was first noted in (Thomsen, 1995) for CHOCS.

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

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

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. $\square$

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 right hand side can perform more τ-moves than the left hand side. 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 μCML, we observe that μCML only allows $\oplus$ to be used on *guarded expressions*, and not on arbitrary expressions. We can thus ignore the initial τ-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 τ-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:

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

and:

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

Let $(\approx^n, \approx^s)$ be the largest hereditary bisimulation. Note that we require $\mathcal{R}^s$ to be structure-preserving because it is used to compare the labels in transitions, which may contain abstractions or guarded events.

In the operational semantics of μCML expressions, guarded expressions can only appear in labels, and not as the residuals of transitions. This explains why in the definition of $\approx^n$ labels are compared with respect to the sensitive relation $\approx^s$ whereas the insensitive relation is used for the residuals. For example, if $ge_1 \approx^n \not\approx^s ge_2$ 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 τ-step. On the other hand:

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

since $[\,]$ is precisely the construct which allows us to embed ge_1 and ge_2 in a $\oplus$ context.

Theorem 3.7
$\approx^s$ is a congruence for μCML$^+$, and $\approx^n$ is a congruence for μCML.

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. $\square$

Proposition 3.8
The equivalences on μCML$^+$ have the following strict inclusions:

$$\begin{array}{ccccccc} \sim^1 & \subset & & & & & \approx^1 \\ \cap\downarrow & & & & & & \cap\downarrow \\ \sim^h & \subset & \approx^i & \subset & \approx^s & \subset & \approx^n \\ & & & & \cap\downarrow & & \cap\downarrow \\ & & & & =^h & \subset & \approx^h \end{array}$$

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:

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

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

where:

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

(Note that this settles an open question (Thomsen, 1995) as to whether $\approx^i$ is the largest congruence contained in $\approx^h$.) $\square$

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 τ-moves. The only μCML$^+$ constructor for guarded expressions which allows this is **A**, and in turn occurrences of this can only be generated by the μCML constructor always. Therefore:

Proposition 3.9
For the subset of μCML$^+$ without always and **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 μCML$^+$ without always and **A**, define $\mathcal{R}^s$ 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 **A** can perform a τ-move, and since the only initial moves of $v_i\, w$ are β-reductions, we can show that $(\approx^h, \mathcal{R}^s)$ forms an hereditary bisimulation, and so $\approx^h \subseteq \approx^n$. From this it is routine to show that $\approx^s = =^h$. $\square$

Unfortunately we have not been able to show that $\approx^n$ is the largest μCML 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 μCML contexts.

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

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

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

We will now show that $(\mathcal{R}^n, \mathcal{R}^s)$ forms an 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 $\mathcal{R}^{sl} = \mathcal{R}^l$.

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

$$\begin{array}{ccc} ge_1 & \mathcal{R}^s & ge_2 \\ {\scriptstyle l_1}\downarrow & & \\ e_1 & & \end{array}$$

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

$$\begin{array}{ccc} \mathsf{choose}([ge_1], \mathsf{receive}\,k) & \mathcal{R} & \mathsf{choose}([ge_2], \mathsf{receive}\,k) \\ {\scriptstyle \surd[ge_1 \oplus k?]}\Downarrow & & {\scriptstyle \surd[ge_2 \oplus k?]}\Downarrow \\ \Lambda & \mathcal{R} & \Lambda \end{array}$$

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

$$\begin{array}{ccc} ge_1 \oplus k? & \mathcal{R} & ge_2 \oplus k? \\ {\scriptstyle l_1}\downarrow & & \\ e_1 & & \end{array}$$

which can be completed as:

$$\begin{array}{ccc} ge_1 \oplus k? & \mathcal{R} & ge_2 \oplus k? \\ {\scriptstyle l_1}\downarrow & & {\scriptstyle \hat{l}_2}\Downarrow \\ e_1 & \mathcal{R} & e_2 \end{array} \quad \text{where } l_1\ \mathcal{R}^l\ l_2$$

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

$$\begin{array}{ccc} ge_1 & \mathcal{R}^s & ge_2 \\ {\scriptstyle l_1}\downarrow & & {\scriptstyle l_2}\Downarrow \\ e_1 & \mathcal{R} & e_2 \end{array} \quad \text{where } l_1\ \mathcal{R}^{sl}\ l_2$$

The other cases are simpler, and so $(\mathcal{R}^n, \mathcal{R}^s)$ is an 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 μCML contexts. $\square$

This Theorem should be contrasted with the case of CCS. In (Milner, 1989) section 7.2 it is shown that the largest congruence contained in weak bisimulation is not itself a weak bisimulation.

4 Bisimulation as a congruence

To serve as the basis of a useful semantic theory of μCML, 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 μCML^+, and that $\approx^n$ is a congruence for μCML.

Unfortunately, this proof is not straightforward, due to the higher-order nature of hereditary bisimulation. The problem is not unique to μCML, and it occurs in many higher-order languages, for example typed λ-calculi (Gordon, 1995), the untyped λ-calculus (Howe, 1989), and the Calculus of Higher-Order Communicating Systems (CHOCS) (Thomsen, 1995) .

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 (Thomsen, 1995, Prop. 6.6) adapts Milner's technique (Milner, 1989, 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 (Howe, 1989).

We shall apply Howe's technique to show that $\approx^s$ is a congruence for μCML^+, and that $\approx^n$ is a congruence for μCML^+ 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 (Howe, 1989) and the notation of (Gordon, 1995), 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:

$$\begin{array}{rcl} C & ::= & \cdot_i \mid e \mid cC \mid \mathsf{if}\, C\, \mathsf{then}\, C\, \mathsf{else}\, C \mid (C,C) \mid \mathsf{let}\, x = C\, \mathsf{in}\, C \\ & & \mid CC \mid \mathsf{fix}(x = \mathsf{fn}\ y \Rightarrow C) \mid \langle C,C \rangle \\ & & \mid [C] \mid C!C \mid C? \mid C \Rightarrow C \mid C \oplus C \mid \mathbf{A}C \mid C \Vdash C \end{array}$$

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 a *congruence* iff $e_i \ \mathcal{R}\ f_i$ implies $C[\vec{e}]\ \mathcal{R}\ C[\vec{f}]$.

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

$$\begin{array}{rcl} C_n & ::= & \cdot_i \mid e \mid cC_n \mid \mathsf{if}\, C_n\, \mathsf{then}\, C_n\, \mathsf{else}\, C_n \mid (C_n, C_n) \mid \mathsf{let}\, x = C_n\, \mathsf{in}\, C_n \\ & & \mid C_n C_n \mid \mathsf{fix}(x = \mathsf{fn}\ y \Rightarrow C_n) \mid \langle C_n, C_n \rangle \\ & & \mid C_n!C_n \mid C_n? \mid C_n \Rightarrow C_n \mid \mathbf{A}C_n \mid C_n \Vdash C_n \end{array}$$

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 μCML context is an uneventful context, and so any uneventful congruence is a

congruence for μCML. 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:

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

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:

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

This definition is rather different from Howe's and Gordon's definition of $\widehat{\mathcal{R}} = \{(D[\vec{e}], D[\vec{f}]) \mid e_i\ \mathcal{R}\ f_i\}$. 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 $\mathsf{fix}(x = \mathsf{fn}\,y \Rightarrow e)\ \widehat{\mathcal{R}}^s\ \mathsf{fix}(x = \mathsf{fn}\,y \Rightarrow f)$ when $e\ \mathcal{R}^n\ f$.

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 (Gordon, 1995; Howe, 1989). Show by induction on C that if $e_i\ \mathcal{R}^s\ f_i$ then $C[\vec{e}]\ \mathcal{R}^s\ C[\vec{f}]$. Either $C = \cdot_i$, in which case the result is immediate, or $C = D[\vec{C}]$ and by induction $C_i[\vec{e}]\ \mathcal{R}^s\ C_i[\vec{f}]$, so by definition $C[\vec{e}] = D[\vec{C}[\vec{e}]]\ \widehat{\mathcal{R}}^s\ D[\vec{C}[\vec{f}]] = C[\vec{f}]$. 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:

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

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 (Gordon, 1995).

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

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 $\mathcal{R}^\bullet \subseteq \mathcal{S}$. $\square$

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 $v\ \mathcal{R}^{\bullet s}\ w$:

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

Proof
A variant of the proof in (Gordon, 1995; Howe, 1989). To prove the first part, we proceed by induction on e.

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

The proof of the second part is similar. $\square$

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 an hereditary simulation.

Proposition 4.4
When restricted to closed expressions of μCML$^+$, $\approx^\bullet$ is an hereditary simulation.

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

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

If:

$$\begin{array}{ccc} e & \approx^{\bullet n} & f \\ {\scriptstyle l_1}\downarrow & & \\ e' & & \end{array}$$

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

$$\begin{array}{ccc} e & \approx^{\bullet n} & f \\ {\scriptstyle l_1}\downarrow & & {\scriptstyle \hat{l_2}}\Downarrow \\ e' & \approx^{\bullet n} & f' \end{array}$$

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:

$$\begin{array}{ccccc} e & = & \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 \widehat{\approx^\bullet}^n \mathsf{let}\, x = g_1 \,\mathsf{in}\, g_2 \approx^n & & f \\ {\scriptstyle \tau}\downarrow & & {\scriptstyle \tau}\downarrow & & \\ e' & = & e_1' \uplus e_2[v/x] & & \end{array}$$

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

$$\begin{array}{ccccc} e & = & \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 \widehat{\approx^\bullet}^n \mathsf{let}\, x = g_1 \,\mathsf{in}\, g_2 \approx^n & & f \\ {\scriptstyle \tau}\downarrow & & {\scriptstyle \tau}\downarrow \quad {\scriptstyle \tau}\Downarrow & & {\scriptstyle \varepsilon}\Downarrow \\ e' & = & e_1' \uplus e_2[v/x] \approx^{\bullet n} g_1' \uplus g_2[w/x] \approx^n & & f' \end{array}$$

- if we have:

$$\begin{array}{ccccc} e & = & e_1\, e_2 & \widehat{\approx^\bullet}^n \quad g_1\, g_2 \quad \approx^n & f \\ {\scriptstyle \tau}\downarrow & & {\scriptstyle \tau}\downarrow & & \\ e' & = & e_1' \uplus \mathsf{let}\, y = e_2 \,\mathsf{in}\, e_3[v/x] & & \end{array}$$

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, $e_3[v/x] \approx^{\bullet n} h_3[v'/x] \approx^{n\circ} v'y \approx^{n\circ} wy \approx^{n\circ} g_3[w/x]$, and so:

$$\begin{array}{ccccccc}
e & = & e_1\, e_2 & \widehat{\approx^{\bullet}}^{n} & g_1\, g_2 & \approx^{n} & f \\
\tau\downarrow & & \tau\downarrow & & \tau\Downarrow & & \varepsilon\Downarrow \\
e' & = & e_1' \uplus \mathsf{let}\, y = e_2\, \mathsf{in}\, e_3[v/x] & \approx^{\bullet n} & g_1' \uplus \mathsf{let}\, y = g_2\, \mathsf{in}\, g_3[w/x] & \approx^{n} & f'
\end{array}$$

- if we have:

$$\begin{array}{ccccc}
e & = & \mathsf{fix}(x = \mathsf{fn}\ y \Rightarrow e_1) & \widehat{\approx^{\bullet}}^{n} & \mathsf{fix}(x = \mathsf{fn}\ y \Rightarrow g_1) \approx^{n} \quad f \\
\surd e\downarrow & & \surd e\downarrow & & \\
e' & = & \Lambda & &
\end{array}$$

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

$$\begin{array}{ccccccc}
e & = & \mathsf{fix}(x = \mathsf{fn}\ y \Rightarrow e_1) & \widehat{\approx^{\bullet}}^{n} & \mathsf{fix}(x = \mathsf{fn}\ y \Rightarrow g_1) & \approx^{n} & f \\
\surd e\downarrow & & \surd e\downarrow & & \surd v\Downarrow & & \surd w\Downarrow \\
e' & = & \Lambda & \approx^{\bullet n} & \Lambda & \approx^{n} & f'
\end{array}$$

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

Thus $\approx^{\bullet}$ is an 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, 1992) (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 (Howe, 1992).

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. $\square$

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

Proposition 4.6
When restricted to closed expressions of μCML$^+$, $\approx^{\bullet *}$ is an hereditary bisimulation.

Proof
By Proposition 4.4, $\approx^\bullet$ is an hereditary simulation, and so $\approx^{\bullet *}$ is an hereditary simulation. By Proposition 4.5, $\approx^\bullet$ is symmetric, and so $\approx^\bullet$ is an hereditary bisimulation. $\square$

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 an 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. $\square$

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 μCML$^+$. Although much remains to be done on elaborating the algebraic theory of μCML 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 μCML 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 $\parallel$ and suitable versions of input and output prefixing, (Milner, 1989). The prefixes in μCML^{cv} have a slightly unusual syntax—their equivalents in CCS are given as:

CCS prefix	*μCML^{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 $\parallel$ act like choice and parallel operators from a process algebras

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

$$\begin{array}{rcl} \Lambda \parallel e & \sim^1 & e \\ (e_1 \parallel e_2) \parallel e_3 & \sim^1 & e_1 \parallel (e_2 \parallel e_3) \\ (e_1 \parallel e_2) \parallel e_3 & \sim^1 & (e_2 \parallel e_1) \parallel e_3 \end{array}$$

Thus $\parallel$ 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:

$$v \Vert e \quad \sim^1 \quad e$$

For example:

$$1 \Vert \Lambda \sim^1 \Lambda \qquad \Lambda \Vert 1 \sim^1 1$$

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

$$(\Vert_i e_i) \Vert f$$

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 μCML$^+$ also satisfies the expected laws from process algebras, those of a commutative monoid, although it can only be applied to guarded expressions:

$$\begin{array}{rcl} \Lambda \oplus ge & \sim^1 & ge \\ (ge_1 \oplus ge_2) \oplus ge_3 & \sim^1 & ge_1 \oplus (ge_2 \oplus ge_3) \\ ge_1 \oplus ge_2 & \sim^1 & ge_2 \oplus ge_1 \end{array}$$

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, (Milner, 1989). We can find bisimulations for the following (and hence they are sensitive bisimilar):

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

From this, we can show, by structural induction on syntax 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 μCML viewed as a functional language. One would not expect β-reduction in its full generality in a language with side-effects such as μCML but we do obtain an appropriate call-by-value version:

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

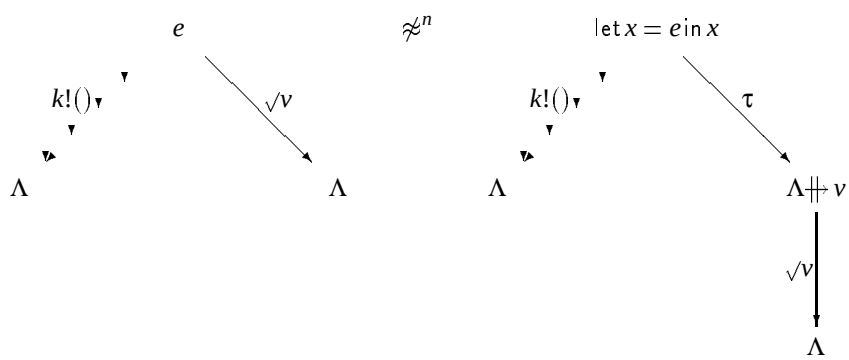
We also have expected laws such as:

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

The last two equations are of particular interest, since they are exactly the left unit and associativity axioms of the monadic metalanguage (Moggi, 1991). 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:

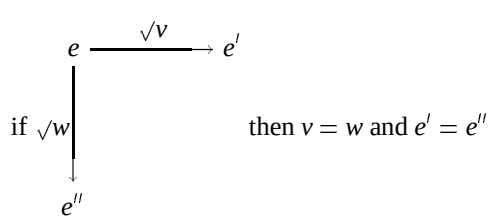


(This is the same example which makes *SKIP* not act as a right unit for sequential composition in CSP (Hoare, 1985) and **exit** not act as a right unit for $\gg$ in LOTOS (ISO 8807, 1989).) Fortunately, we can show that our operational semantics for μCML satisfies four properties which allow us to 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:



It is *forward commutative* iff:

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

Note that since α may be an input move, e'' may be an open term, so we need to take the open extension $\xrightarrow{\surd v^{\circ}}$ of the termination relation.

It is *backward commutative* iff:

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

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, μCML does satisfy these conditions.

Proposition 5.1
μCML satisfies single-valuedness, value determinacy, forward commutativity and backward commutativity.

Proof
A routine induction on syntax. $\square$

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. $\square$

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 μCML 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 μCML 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 (Moggi, 1991).

6 Comparing μCML$^+$ and λ_{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 a formal connection between this semantics and the reduction semantics for λ_{cv} presented in (Reppy, 1992). We have not considered λ_{cv} in its entirety and so the comparison will be confined to the common subset, namely μCML^{cv}. We first reproduce, as faithfully as possible, the reduction semantics of Reppy as it applies to μCML. From this reduction semantics we then derive a labelled transition system for μCML 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 λ_{cv} and μCML^{cv}:

- We do not consider the λ_{cv} constructs `guard` and `wrapAbort`. We conjecture that the operational semantics of μCML would need to be considerably altered to cope with translating these constructs.
- We omit the λ_{cv} construct `chan x in e` since we cannot encode unique channel name generation in μCML, although it should not be difficult to add it using operational rules à la π-calculus. However this would require using a bisimulation similar to Sangiorgi's (Sangiorgi, 1992) context bisimulation for the higher-order π-calculus.
- We have added recursive function types to μCML^{cv} because in (Reppy, 1992) recursion is encoded using process creation and unique channel name generation.
- In λ_{cv}, constant functions such as `wrap` are values, where in μCML they have to be coded as $(\mathsf{fn}\ x \Rightarrow \mathsf{wrap}\ x)$. This restriction has no effect on the expressive power of μCML, and makes it simpler to reason about the operational semantics, since any value of type $A \rightarrow B$ must be of the form $\mathsf{fix}(x = \mathsf{fn}\ y \Rightarrow e)$.

We now present Reppy's reduction semantics for μCML^{cv}. In (Reppy, 1992) this is represented by a transition relation between multi-sets of μCML^{cv}, or more generally λ_{cv}

expressions. Instead of multi-sets we use *configurations* of μCML^{cv} expressions given by the grammar:

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

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

The semantics of (Reppy, 1992) is expressed as a reduction relation $\Longrightarrow$ between configurations and reductions have four independent sources. The first involves a sequential reduction within an individual μCML 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 & Felleisen, 1991), and the contexts that permit reduction are given by the following grammar:

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

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

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

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 (Reppy, 1992), 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 $e \longmapsto e'$. $\square$

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 any k the relation:

$$ge \overset{k}{\bowtie} ge' \ \textbf{with}\ (e,e')$$

read as "ge matches ge' on k with result (e,e')" is defined to be the least relation satisfying the rules in Figure 6a. Intuitively this means that two concurrent threads e_1, e_2 of the form

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

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

$$\frac{ge \overset{k}{\bowtie} ge'\ \textbf{with}\ (e,e')}{ge' \overset{k}{\bowtie} ge\ \textbf{with}\ (e',e)}$$

Figure 6a. 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 6b. The rules for immediate evaluation of events

$e_1 = E_1[\mathsf{sync}[ge]], e_2 = E_2[\mathsf{sync}[ge']]$ 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 6b.

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

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

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

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

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

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

Theorem 6.2
The μCML^{cv} semantics of a configuration is weakly bisimilar to its μCML$^+$ 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 μCML^{cv}, when modelled in the μCML$^+$ semantics, require in addition some 'housekeeping' reductions. A typical example is the reduction:

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

In μCML$^+$ 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 μCML$^+$ 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 spawn; expressions in μCML$^+$ 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 μCML$^+$ and μCML^{cv} are:

$$\begin{array}{lcl} (\mathsf{spawn}\ v, e) & \xrightarrow{\tau} & (\Lambda \Vert v\,() \Vert (), e) \\ (\mathsf{spawn}\ v, e) & \longmapsto^{\tau} & v\,() \Vert ((), e) \end{array}$$

This difference is handled by working with the μCML^{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:

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

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

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. $\square$

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

Proposition 6.4

$$\begin{array}{ccc} e & \xrightarrow{\tau_H^*} & e' \\ \text{If } l \downarrow & & \\ e'' & & \end{array} \qquad \begin{array}{ccc} e & \xrightarrow{\tau_H^*} & e' \\ \text{then } l \downarrow & & \downarrow \hat{l} \\ e'' & \xrightarrow{\tau_H^{*\circ}} & e''' \end{array}$$

Proof
First show by induction on ge that $ge \not\xrightarrow{\tau_H}$. 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. $\square$

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

Proposition 6.5
For any μCML$^+$ 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. $\square$

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

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

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:

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

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

$$\begin{array}{ccc} e_1 & \mathcal{R} & e_2 \\ \tau_H \downarrow & & \\ e_1' & & \end{array} \quad \text{as} \quad \begin{array}{ccc} e_1 & \mathcal{R} & e_2 \\ \tau_H \downarrow & & \tau_H \downarrow \\ e_1' & \mathcal{R} & e_2' \end{array}$$

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 μCML^{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:

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

and:

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

Proof
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 μCML^{cv} semantics of a configuration is weakly bisimilar to its μCML^{+} semantics.

Proof
Intuitively we know, from Proposition 6.3, that μCML^{+} expressions are semantically equivalent to their tidy forms, and Proposition 6.7 can be used to transform μCML^{cv} moves from an expression into μCML^{+} 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:

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

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:

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

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:

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

The result follows. $\square$

7 Conclusions

In this paper we have defined a compositional operational semantics for a core subset of CML, called μCML, 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 μCML^+ although it is worth pointing out that all of the added constructs can be defined in the core language μCML up to weak bisimulation equivalence.

Much research remains to be done. The algebraic theory of μCML, 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*, (Sangiorgi, 1992).

As pointed out in Section 3 our semantic equivalence, $\approx^n$, is based on the *late* version of bisimulations, (Milner *et al.*, 1992). 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 μCML contexts.

In Section 3, we were forced to develop the theory of hereditary bisimulations because of the usual problems of τ 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 (Ferreira, 1995). 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 (Jeffrey, 1995) that the resulting semantics can be presented in terms of computational monads (Moggi, 1991).

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, (Nielson & Nielson, 1993; Nielson & Nielson, 1996). 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 (Reppy, 1991b; Reppy, 1992; Panangaden & Reppy, 1996). This semantics has been used in (Berry *et al.*, 1992) 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 (Bolignano & Debabi, 1994; Debabi, 1994) 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 (Havelund, 1994; Baeten & Vaandrager, 1992) 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 (Ferreira & Hennessy, 1995) to give an operational semantics of a language which can be considered to be an untyped version of μCML. However bisimulation based equivalences

are not developed in (Ferreira & Hennessy, 1995); instead a testing equivalence is defined (Hennessy, 1988) and a fully-abstract denotational semantics based on Acceptance Trees is given.

Other languages which contain much in common with CML include *CHOCS* (Thomsen, 1995), *FACILE* (Giacalone *et al.*, 1989), *PICT* (Pierce & Turner, 1995), *ACTORS* (Agha *et al.*, 1994) and *HO*π (Sangiorgi, 1992). 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 (Thomsen, 1995) 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 μCML to include channel generation it will be necessary to adopt the *context bisimulation equivalence*, originally developed in (Sangiorgi, 1992). 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 $\mu\mathrm{CML}^+$, 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:

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

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

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

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

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. $\square$

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

$$\begin{array}{ccc} C & \xrightarrow{\tau_H^*} & e \\ {\scriptstyle l}\downarrow & & \\ C' & & \end{array} \quad \text{as} \quad \begin{array}{ccc} C & \xrightarrow{\tau_H^*} & e \\ {\scriptstyle l}\downarrow & & \downarrow{\scriptstyle l} \\ C' & \xrightarrow{\tau_H^{*\circ}} & \equiv e' \end{array}$$

and:

$$\begin{array}{ccc} C & \xrightarrow{\tau_H^*} & e \\ & & \downarrow{\scriptstyle l} \\ & & e' \end{array} \quad \text{as} \quad \begin{array}{ccc} C & \xrightarrow{\tau_H^*} & e \\ {\scriptstyle l}\downarrow & & \downarrow{\scriptstyle l} \\ C' & \xrightarrow{\tau_H^{*\circ}} & \equiv e' \end{array}$$

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

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

$$\begin{array}{ccccccc} C & = & E_1^+[E_2[cv]] & \xrightarrow{\tau_H^*} & E_1^+[E_2^+[cv]] & = & e \\ {\scriptstyle \tau}\downarrow & & {\scriptstyle \tau}\downarrow & & & & \downarrow{\scriptstyle \tau} \\ C' & = & E_1^+[E_2[\delta(c,v)]] & \xrightarrow{\tau_H^*} & E_1^+[E_2^+[\delta(cv)]] & \equiv & E_1^+[E_2^+[\Lambda \Vdash \delta(cv)]] \end{array}$$

- If $C \xmapsto{\surd v} C'$ then $C = C'' \Vdash v$ and $C' = C'' \Vdash \Lambda$, so:

$$\begin{array}{ccccccc} C & = & C'' \Vdash v & \xrightarrow{\tau_H^*} & e'' \Vdash v & = & e \\ {\scriptstyle \surd v}\downarrow & & {\scriptstyle \surd v}\downarrow & & & & \downarrow{\scriptstyle \surd v} \\ C' & = & C'' \Vdash \Lambda & \xrightarrow{\tau_H^*} & e'' \Vdash \Lambda & = & e'' \Vdash \Lambda \end{array}$$

- If $C \xmapsto{k!v} C'$ then (from the definition of $C \xmapsto{k!v} C'$ 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 Proposi-

tions A.1 and A.3:

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

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 $\mu\mathrm{CML}^+$ and $\mu\mathrm{CML}^{cv}$, $e = E[g]$, $f \xrightarrow{\tau_H}{}^* g$, $e' = E[g']$ and $g \xrightarrow{\alpha} g'$ then by induction $f \xmapsto{l} C' \xrightarrow{\tau_H}{}^* f' \equiv g'$ and we can show by induction on E that $E[g] \xmapsto{l} \equiv E[C]$ so by Propositions 6.6:

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

Otherwise:

- If $C = c\,f$ then $f \xrightarrow{\tau_H}{}^* g$, g is tidy and $c\,g \xrightarrow{\tau_H}{}^* e$, so either:

  — $c = \mathsf{sync}$, $e = g' \Vert ge$, $g \xrightarrow{\surd[ge]} g'$, and $f \xrightarrow{\tau_H}{}^* g$, so by induction and the definition of $\xmapsto{\surd v}$, $f = g = [ge]$ and $g' = \Lambda$, so $e' = \Lambda \Vert g''$ and $ge \xrightarrow{\alpha} g''$, so by Proposition A.5, $\mathsf{sync}[ge] \xmapsto{\alpha} g''$, and so:

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

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

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

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

$$\begin{array}{ccccccc}
C & = & cv & \xrightarrow{\tau_H^*} & \mathsf{spawn}\, g & = & e \\
\tau \downarrow & & & & \downarrow \tau & & \downarrow \tau \\
\delta(c,v) & \xrightarrow{\tau_H^{*\circ}} & \delta(c,v) & = & \Lambda \Vert \delta(c,v) & = & e'
\end{array}$$

- If $C = f_1\, f_2$ then $f_1 \xrightarrow{\tau_H,*} g_1 \xrightarrow{\surd v} g_1'$ where $v = \mathsf{fix}(x = \mathsf{fn}\, y \Rightarrow g_3)$, $f_2 \xrightarrow{\tau_H,*} g_2$, $e = g_1' \Vert \mathsf{let}\, y = g_2\, \mathsf{in}\, g_3[v/x]$, so by induction and the definition of $\xrightarrow{\surd v}$, $f_1 = g_1 = v$ and $g_1' = \Lambda$, and so either:

  — $e' = g_1' \Vert \mathsf{let}\, y = g_2'\, \mathsf{in}\, g_3[v/x]$ and $g_2 \xrightarrow{\alpha} g_2'$ so by induction (up to associativity of $\Vert$ and Λ being a left unit), $f_2 \xrightarrow{\alpha} C' \Vert f_2' \xrightarrow{\tau_H,*\circ} f_3 \Vert f_2'' \equiv g_2'$, and so:

$$\begin{array}{ccccccc}
C & = & v\, f_2 & \xrightarrow{\tau_H^*} & \Lambda \Vert \mathsf{let}\, y = g_2\, \mathsf{in}\, g_3[v/x] & = & e \\
\alpha \downarrow & & & & \downarrow \alpha & & \downarrow \alpha \\
C' \Vert v\, f_2' & \xrightarrow{\tau_H^{*\circ}} & f_3 \Vert \mathsf{let}\, y = f_2''\, \mathsf{in}\, g_3[v/x] & \equiv & \Lambda \Vert \mathsf{let}\, y = g_2'\, \mathsf{in}\, g_3[v/x] & = & e'
\end{array}$$

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

$$\begin{array}{ccccccc}
C & = & vw & \xrightarrow{\tau_H^*} & \Lambda \Vert \mathsf{let}\, y = w\, \mathsf{in}\, g_3[v/x] & = & e \\
\tau \downarrow & & & & \downarrow \tau & & \downarrow \tau \\
g_3[v/x][w/y] & \xrightarrow{\tau_H^*} & g_3[v/x][w/y] & = & \Lambda \Vert \Lambda \Vert g_3[v/x][w/y] & = & e'
\end{array}$$

The result follows. $\square$

References

Agha, G., Mason, I., Smith, S., & Talcott, C. (1994). A foundation for actor computation. *J. functional programming*.

Baeten, J. C. M., & Vaandrager, F. W. (1992). An algebra for process creation. *Acta informatica*, **29**(4), 303–334.

Bergstra, J. A., & Klop, J. W. (1985). Algebra of communicating processes with abstraction. *Theoret. comput. sci.*, **37**, 77–121.

Berry, D., Milner, R., & Turner, David N. (1992). A semantics for ML concurrency primitives. *Proc. popl 92*.

Bolignano, D., & Debabi, M. (1994). A semantic theory for concurrent ML. *Proc. TACS '94*.

Debabi, M. (1994). *Integration de paradigmes de programmation paralle, fonctionnelle et imperative*. Ph.D thesis, Universite D'Orsay.

Ferreira, W. (1995). *Semantic theories for concurrent ML*. D.Phil thesis, COGS, Sussex Univ.

Ferreira, W., & Hennessy, M. (1995). Towards a semantic theory of CML. *Proc. MFCS 95*. Lecture Notes in Comp. Sci., no. 969. Springer-Verlag.

Giacalone, A., Mishra, P., & Prasad, S. (1989). Facile: A symmetric integration of concurrent and functional programming. *Pages 184–209 of: Proc. Tapsoft 89*. LNCS, vol. 352. Springer-Verlag.

Gordon, A.D. (1995). Bisimilarity as a theory of functional programming. *Proc. MFPS 95*. Electronic Notes in Comp. Sci., no. 1. Springer-Verlag.

Gunter, C. (1992). *Semantics of programming languages*. MIT Press.

Havelund, K. (1994). *The fork calculus: Towards a logic for concurrent ML*. Ph.D thesis, École Normale Superieur, Paris.

Hennessy, M. (1988). *Algebraic theory of processes*. MIT Press.

Hoare, C. A. R. (1985). *Communicating sequential processes*. Prentice-Hall.

Holmström, S. (1983). PFL: A functional language for parallel programming. *Pages 114–139 of: Proc. declarative programming workshop*.

Howe, D. (1989). Equality in lazy computation systems. *Pages 198–203 of: Proc. LICS 89*.

Howe, D. (1992). *Proving congruence of simulation orderings in functional languages*. Unpublished manuscript.

ISO 8807. (1989). *LOTOS—a formal description technique based on the temporal ordering of observational behaviour*.

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

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

Milner, R. (1991). The polyadic π-calculus: a tutorial. *Proc. international summer school on logic and algebra of specification*.

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

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

Nielson, F., & Nielson, H. R. (1993). *From CML to process algebras*. Report DAIMI FN-19. Dept. Comp. Sci., Aarhus University.

Nielson, F., & Nielson, H. R. (1996). Communication analysis for concurrent ML. In *ML with concurrency*, Springer.

Nikhil, S. (1990). *Id reference manual*. MIT Lab. for Comp. Sci.

Pierce, B. C., & Turner, D. N. (1995). *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.

Plotkin, G. (1975). Call-by-name, call-by-value and the lambda-calculus. *Theoret. comput. sci.*, **1**, 125–159.

Reppy, J. (1991a). A higher-order concurrent langauge. *Pages 294–305 of: Proc. SIGPLAN 91*.

Reppy, J. (1991b). *An operational semantics of first-class synchronous operations*. Technical report TR 91-1232. Dept. Comp. Sci., Cornell Univ.

Reppy, J. (1992). *Higher-order concurrency*. Ph.D thesis, Cornell Univ.

Panangaden, P., & Reppy, J. (1996). The essence of concurrent ML. In *ML with concurrency*, Springer.

Sangiorgi, D. (1992). *Expressing mobility in process algebras: First-order and higher-order paradigms*. Ph.D thesis, LFCS, Edinburgh Univ.

Thomsen, B. (1995). A theory of higher order communicating systems. *Inform. and comput.*, **116**(1), 38–57.

Wright, A., & Felleisen, M. (1991). *A syntactic approach to type soundness*. Technical report TR91-160. Dept. of Comp. Sci., Rice Univ.